Equational Reasoning about Object-Oriented Programs

Md. Nour Hossain

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of

Master of Science

Faculty of Mathematics and Science, Brock University
St. Catharines, Ontario

To Professor Michael Winter

&

A. Hossain Rana

Abstract

Formal verification of software can be an enormous task. This fact brought some software engineers to claim that formal verification is not feasible in practice. One possible method of supporting the verification process is a programming language that provides powerful abstraction mechanisms combined with intensive reuse of code. In this thesis we present a strongly typed functional object-oriented programming language. This language features type operators of arbitrary kind corresponding to so-called type protocols. Subclassing and inheritance is based on higher-order matching, i.e., utilizes type protocols as basic tool for reuse of code. We define the operational and axiomatic semantics of this language formally. The latter is the basis of the interactive proof assistant VOOP (**V**erified **O**bject-**O**riented **P**rograms) that allows the user to prove equational properties of programs interactively.

Acknowledgements

I owe my deepest gratitude to my supervisor, Professor Michael Winter, for many insightful conversations during the development of the ideas in this thesis, and for helpful comments on the text. Without his encouragement, guidance and support from the initial to the final level this thesis would not have been possible.

After my parent, my brother Amir Hossain Rana has been an inspiration throughout my life. I would like to show my gratitude to him for all he is, and all he has done for me.

For financial support, I thank the Department of Computer Science, Brock University and my brother Rana.

Lastly, I offer my regards and blessings to all of those who supported me in any respect during the completion of this thesis.

M.N.H

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are safety-critical situations in which it is highly desirable to be sure that a program behaves properly, i.e., as intended. Intensive testing might help to achieve that goal by detecting some errors in the program. However, this method will never be able to ensure that the program is error free. A proof is a logical or mathematical argument showing that a certain property holds in all circumstances. Program correctness and formal verification applies logic and mathematical proofs to properties of programs. Therefore, this method guarantees that the program will behave as required whatever the conditions.

A common slogan of some software engineers is that formal verification of programs is not feasible in practice. The main reason for this problem is that most of the existing tools and methods are based on widely used programming languages such as C, C++, Java, Ada or Pascal. These languages usually offer only a few (or sometimes no) abstraction and inheritance mechanisms. For example, type declarations in these languages are based on concepts naturally supported by a computer such as arrays and pointers, and not based on abstract type concepts such as product, sum, recursion, and self types. Similarly, inheritance (if available) is based on subtyping – sometimes even on restricted versions of that. It is known that this limits severely opportunities for reusing code. Therefore, one is forced to start on a very basic and machine dependent level, which generates a huge amount of proof obligations. In real world applications proving all obligations seems impossible, or is at least very time consuming and, hence, expensive.

Our approach is based on a functional object-oriented programming language that offers powerful abstraction and inheritance mechanisms together with a proof calculus that allows inheritance of properties and proofs.

Programs are supposed to be written in this language, verified using the calculus, and then, if needed, translated into a program of a standard programming language.

1.1 Program Verification Technique

Program verification is the software part of formal verification. Formal verification applies logic and mathematical technique depending on formal specification, i.e., properties of program, to state the correctness of a system, i.e., program. A property of program means the mathematical representation of the implementation, which concentrates on what the program should do rather than how the program should do [19, 20, 21].

Different methods for verification have been developed. The most prominent approach is based on Floyd-Hoare logic [22, 23]. This logic is based on so-called Hoare triples consisting of a precondition, a program, and a post condition. The meaning of such a triple can be summarized as follows. If the precondition is true and the program terminates, then the post condition will be true. Since termination of the program is assumed rather than shown such a property is called a partial correctness property. This logic is normally used for imperative programming languages, and axioms and proof rules for various construction in such a language have been provided.

Functional programming languages are not based on state-based. Because of this reasoning in such a language is normally based on equations. This style of reasoning is completely different form the Floyd-Hoare logic. Here the axioms and proof rules are based on algebraic equations, and reasoning is similar to regular algebras known from high school. Since our object-oriented language has a functional kernel we have adopted the same style for formal verification.

Now we are going to show how to reason functional programming language (Haskell) by proving one property of that program. The method we are going to use to prove is called method of induction. According to [18]" In order to prove that a logical property P(xs) holds for all finite lists xs we have to do two things.

1. Base case: Prove P([]) outright.

2. Induction step: Prove P(x:xs) on the assumption that P(xs) holds. In another words P(xs)⇒P(x:xs) has to be proved. The P(xs) here is called the induction hypothesis since it is assumed in proving P(x:xs)".

Notice that in Haskell [] denotes the empty list and x:xs a list with head x and tail xs.

Two Haskell functions sum and doubleAll are defined and declared below. The function sum () receives list of integers and return the sum of the list of integers. The other function takes a list of integers and returns a list by doubling all the elements of the receiving list. Functions in Haskell are usually defined using pattern matching. In our case of lists this means that we have one section of code that defines the function for the empty list and one another section of code that defines the function for non-empty lists, i.e., lists x:xs with a head x and a tail xs.

1. sum : [Int] -> Int
2. sum [] =0
3. sum (x:xs)= x + sum(xs)
4. doubleAll : [Int] -> [Int]
5. doubleAll [] = []
6. doubleAll (z:zs) = 2 * z : doubleAll zs.

1.1.1 Theorem

Now we will prove that sum a list after doubling all its elements is same as doubling the sum of a list of elements.

$$sum\ (doubleAll\ xs) \ = \ 2 \ * \ sum\ xs. \tag{1.1}$$

1.1.2 Proof

In order to prove this theorem we will have to prove that,

1. Base: sum (doubleAll []) = 2 * sum [].
2. Induction: sum (doubleAll (x:xs)) = 2 * sum (x:xs). Assuming that the hypothesis is: sum (doubleAll xs) = 2 * sum xs.

Base left-hand side:

sum (doubleAll [])
= sum[] (by 5)
= 0 (by 2)

Base right-hand side:

2 * sum []
= 2*0 (by 2)
= 0 (by arith *)

The Induction Step left-hand side:

sun (doubleAll (x:xs))
= sum (2*x : doubleAl l xs) (by 6)
= 2 *x + sum (doubleAll xs) (by 3)
= 2* x + 2 * sum xs by hypothesis

The Induction Step right-hand side:

2 * sum (x:xs)
= 2 * (x + sum xs) (by 3)
= 2*x + 2 * sum xs (by arith *)

So from above proof it is clear that the two programs sum and double are correct for all finite lists. The above example and proof is collected from [18].

1.2 Some Proof Systems

There are different, i.e., model checking, logical interface, approaches to formal verification. Though in hardware development formal verification is used widely, in software engineering field and industry it is still languishing. A couple of different systems used for functional programming languages are explained very briefly as follows:

HOL (Higher Order Logic):

HOL is a theorem proving system or family which prove theorem by man-machine collaboration. There are four versions of HOL namely HOL4, HOL Light, Isabelle and ProofPower. The programming language used by this HOL family is ML (Meta-Language) and its successors (Moscow ML, OCaml, Standard ML). ML is a functional programming language developed by Robin Milner and others. This in non-pure functional programming language and it also has side effect [24, 25].

ACL2 (A Computational Logic for Applicative Common Lisp):

ACL2 is not only a programming language but also a model prover. By using ACL2 one can model computer systems (software and Hardware) as well as prove properties of those models. There is no side effect for ACL2 programming language but it is untyped. The proof system works on first order logic. The language used to build ACL2 is Common Lisp [26, 27].

Coq :

Coq is an interactive proof management system. Coq produces a dependently typed functional programming language by mechanically checking proofs for mathematical assertions and helping to find formal proofs. The core language Coq consists is called calculus of inductive constructions, which is a modification of the formal language calculus of constructions (CoC) [28, 29].

Some other proof systems are Theorem Proving System and the Educational Theorem Proving System (TPS and ETPS) works with simply-typed lambda calculus, Prototype Verification System (PVS) works with higher order logic, Mizar system works with first order logic and PhoX , a automated theorem proving system works on higher order logic [30, 31, 32, 33, 34].

Chapter 2

Object-Oriented Features

In this chapter we just specified some common features of different object oriented programming languages. A good resource for all of those features is [1], we tried to explain them as brief as possible in this chapter.

2.1 Object Orientation

In [2] the characteristics of object-oriented languages are summarized as follows:

"Object-oriented programming (OOP) is a programming paradigm using *objects*, i.e., data structures consisting of data fields and methods together with their interactions, to design applications and computer programs" [2]. Object-Oriented languages allow reuse of software components better than traditional procedural languages.

We can reuse a module by importing it in several other modules or instantiating it with different parameters. In case of procedural languages, an exact agreement in types or interface is required in order to enjoy this reuse property. But in object-oriented language object replacement and method replacement require only approximate agreement, instead of exact agreement [1].

Various mechanisms allow replacing objects. In general terms, one may replace an object with a new one that has at least the same set of attributes. Any additional attributes of the new object remain as *invisible attributes*; they are preserved but are not directly accessible [1].

The notion of *self* is really important for replacement mechanism because in a method *self* can refer to its host object as well as its sibling methods. The dynamic notion of self allows a method to accomplish a new behaviour when inherited into a derived object, depending on the changes in siblings through inheritance and overriding. It makes the method reuse flexible and expansive [1].

Another important concern of these replacement mechanisms is that methods are inseparable and encapsulated into objects. The flexibility in object and method replacement, as well as existence of invisible attributes in objects makes method extraction from an object and reusing it unsound [1]. But By introducing subprotocol relations and type operators we can make these replacement mechanisms trustworthy.

2.2 General Features

Class-based languages are the common, most developed and popular object oriented programming model. Classes describes the structure and behaviour of objects. An object or instance can be created from a class `c` by using some construction, i.e., `new c`. The special identifier `self` generally refers to the host objects. The fields(data) and procedures or methods of an object are collectively called its attributes. In the following example taken from [1] a class named `cell` is defined as:

```
class cell is
   var contents: Integer := 0;
   method get(): Integer is
     return self.contents;
   end;

  method set(n: Integer) is
     self.contents := n;
   end;
end;
```

The class `cell` describes objects having an integer variable `contents` and two methods named `get` and `set`. The `contents` is initialized to zero. The `get` method has no parameter and it returns the value of `contents`. The `set` method has one parameter and it stores the parameter in the `contents` field.

Classes are inseparable component for the notions of subclasses and inheritance. Subclass is also called child class or derived class. Like any class, a subclass describes the structure of a set of objects by inheriting or overriding its direct superclass. A superclass is also knows as base class, or parent class. Declaration of a subclass `reCell` of class `cell` taken from [1] is as follows:

```
subclass reCell of cell is
   var backup :Integer:= 0;
   override set(n:= Integer) is
      self.backup:= self.contents;
      super.set(n);
   end;
   method restore()is
     self.contents:= self.backup;
   end;
end;
```

The class `reCell` is an extension of our previous class `cell`. It describes objects having an integer variable `backup`, an overridden method `set` and an additional method `restore`. The overridden `set` method makes a backup of the `contents` field into `backup` field before updating it. The `super.set(n)` invokes the old version of `set` from the `cell` class. The `restore` method restores the `contents` to its previous value.

Inheritance is a technique of reusing attributes from a superclass to its subclasses. If a class inherits from another class, then for sure the inheriting class is a subclass of the inherited class but not the vice versa. In a class declaration an occurrence of `self` always refers to an object of that class. In a method that a subclass inherits from a superclass, `self` refers to an object of the subclass , not an object of the superclass.

An important concept in object-oriented programming languages is the subtype relationship. The subtype relation <: itself is a reflexive and transitive relation on the types of objects. We do not give a precise definition of this relation at this point. However, the relation satisfies the following property known as *subsumption*:

$$\text{If } a : A \text{ and } A <: B \text{ then } a : B. \tag{2.1}$$

Notice that $a : A$ denotes the fact that object a has type B. This property allows subtype polymorphism, i.e., a kind of polymorphism where programs may accept and return values that are actually of a subtype of the declared type. The type of entities polymorphic in the sense above is denoted by Forall $(X <: A)B$.

The subtype relationship also implies a mechanism of inheritance. Objects of a class c_1 can use code from a class c_2 if the type of objects of c_2 (the instance type of c_2) is a supertype of the type of the objects of c_1 (the instance type of c_1). Actually, in most object-oriented programming languages inheritance is based on this fact. This principle is also known under the slogan *subclassing-is-subtyping / Inheritance-is-subtyping.*

2.3 Variant Notions

According to [1], the definition of covariant, contravariant and invariant are as follows: "The type A×B is the type of pairs with left component of type A and right component of type B. The operation fst(c) and snd(c) extract the left and right components respectively of an element c of type A × B. We say that × is a *covariant* operator, because

$$A \times B <: A' \times B' \text{ provided that } A <: A' \text{ and } B <: B'. \tag{2.2}$$

The type $A \to B$ is the type of functions with argument type A and result type B. We say that $\to$ is a *contravariant* operator in its left argument because $A \to B$ varies in the opposite sense as A; the right argument is instead *covariant*:

$$A \to B <: A' \to B' \text{ provided that } A' <: A \text{ and } B <: B'. \tag{2.3}$$

Let us now consider pairs whose components can be updated having type $A{*}B$. Given p: $A * B$, $a : A$ and $b : B$, we have operator $getLft(p) : A$ and $getRht(p) : B$ that extract components and operations $setLft(p, a)$ and $setRht(p, b)$ that destructively update components. The operator $*$ does not enjoy any *covariance* or *contravariance* properties:

$$A * B <: A' * B' \text{ provided that } A = A' \text{ and } B = B'. \tag{2.4}$$

we say that $*$ is an *invariant* operator".

These properties allow some flexibility in inheritance called *method specialization*, i.e., the actual type of a method in a subclass can actually be different than the type of that method in the superclass as long as the subtype relationship remains valid. However, this is rarely implemented in commonly used object-oriented languages.

2.4 Advanced Features

In various programming languages the objects interfaces are mixed up with implementations. So it is impossible to keep specifications separate from implementations. But by introducing object types (list of attributes and their types) we can achieve this goal. The object type are independent of specific classes, appropriate in interfaces, implemented separately and in more than one way. The object type for a class naming cell taken from [1] is as follows:

```
ObjectType Cell is
   var contents:Integer;
   method get():Integer;
   method set(n:Integer);
end;
```

The subtype relation was previously based on the subclass relation. When object types are independent of classes, we provide an independent definition. For two object types o and o′ we have o′ <: o if o′ has the same components(name of a field or a method and its associated types) as o and possibly more. As a consequence of the independent definition of subtyping, we often have the following relationship between subclassing and subtyping:

$$\text{If } c' \text{ is a subclass of } c, \text{ then } ObjectTypeOf(c') <: ObjecttypeOf(c). \tag{2.5}$$

Subclassing-is-subtyping property is a double implication, but the converse of this new definition does not hold: there may be unrelated c and c′ such that ObjectTypeOf(c)=o and ObjectTypeOf(c′)= o′, with o′ <: o. Subclassing still implies subtyping, so all the previous uses of subsumption are still allowed. But since subsumption is based on subtyping and not subclassing, we now have even more freedom in subsumption.

Another opportunity of flexibility arises when the type of a method contains a recursive occurrence of the instance type of the class itself. Similar to a special variable `self` referring to the object itself, one might introduce a type variable `Self` referring to the type of `self`. This concept is known as *self types*. Method specialization together with self types leads to a more flexible form of inheritance based on subtyping. Nevertheless, even a programming language based on these principles has severe limits. Due to the contravariance of the function type in its parameter, a self type parameter in a function does not lead to a subtype relationship, and, hence, the function cannot be inherited if *subclassing-implies-subtyping* is assumed. Notice that some languages such as Eiffel permit arguments and returns types to be modified covariantly, even though this is theoretically unsound [6, 8].

Consider two classes `maxClass` and `minMaxClass` taken from [1]:

```
class maxClass is
     var n:Integer:=0;
     method max(other:Self):Self is;
        if self.n>other.n then return self
        else return other end;
     end;
end;

subclass minMaxClass of maxClass is
    method min(other:Self):Self is;
       if self.n<other.n then return self
       else return other end;
    end;
    override max (other:Self):Self is
       if other.min(self) = other then return self
       else return other end;
    end;
end;
```

In the above two classes `max`, `min` and overridden `max` are binary methods because they operate on two objects: `self` and `other`. The type of `other` is given by a contravaroant occurrence of `Self` [9].

Any instance of `maxClass` has type `Max` and any instance of `minMaxClass` has type `MinMax`. Although `minMaxClass` is a subclass of `maxClass`, `MinMax` can not be a subtype of `Max`. The type definition of these two types taken from [1] are as follows:

```
ObjectType Max is
  var n:Integer;
  method max(other:Max):Max;
end;
ObjectType MinMax is
   var n: Integer;
   method max(other:MinMax):MinMax;
   method min(other:MinMax):MinMax;
end;
```

In order to verify this claim suppose mm' is an instance of `minMaxClass`, i.e., mm': `MinMax`. If `MinMax` were a subtype of `Max`, then mm'`:Max` and mm'`.max(m)` would be allowed for any `m` of type `Max`. But `m` may not have any `min` attribute, i.e., the overridden `max` in `mm'` of the `minMaxClass` performs an illegal operation. Therefore, the property `MinMax<:Max` does not hold.

2.5 Subprotocol Relation

Even though a subtype relationship is not valid in our previous example it seems intuitively possible to inherit from `MaxClass` into `minMaxClass`. This requires a new relationship between the two classes on which inheritance can be based. Such a relationship is given by a subprotocol relation [7]. In order to find this subprotocol relation, according to [1] two type operators, `MaxProtocol` and `MinMaxProtocol` are as follows:

```
ObjectOperator MaxProtocol[X] is
     var n: Integer;
     method max(other:X):X;
end;

ObjectOperator MinMaxProtocol[X] is
     var n: Integer;
     method max(other:X):X;
     method min(other:X):X;
end;
```

We can apply `Self`, `Max`, `MinMax` or any other type to the above type operators. As an example if we apply the type `MinMax` to them, we will get:

```
ObjectOperator MaxProtocol[MinMax] is
     var n: Integer;
     method max(other:MinMax):MinMax;
end;

ObjectOperator MinMaxProtocol[MinMax] is
     var n: Integer;
     method max(other:MinMax):MinMax;
     method min(other:MinMax):MinMax;
end;
```

Now we can find two formal relationships between Max and MinMax.

- MinMax <: MaxProtocol[MinMax]
- MinMaxprotocol[T]<: MaxProtocol[T] for all types T.

Each property above is basis for a relationship called *matching* [3, 4] between `MinMax` and `Max`. The first version is called *F-bounded matching* and the second *higher-order matching*. Since *F*-bounded matching does not have nice theoretical properties, i.e., it is not transitive [5], higher-order matching is normally chosen as a basis for inheritance.

Chapter 3

The Programming Language

In this chapter we describe in detail the outline of the programming language, its type system, and its operational semantics. The language was inspired by the higher-order object calculus presented in [1].

3.1 Summary of the Features of the Language

The language can be characterized by the following features:

1. It is strongly typed.
2. It has type operators, i.e., type protocols. The type operators are also typed by entities called `Kind`.
3. It support polymorphism, i.e., type can be passed to other functions and returned as the result of a function.
4. The subclassing and inheritance can be encoded in our language using type protocols.
5. It contains features for proving correctness of programs.

3.2 The Syntax of Our Language

The syntax of our language consists of three syntactical components: `Kind`, `Type` and `Program`. They are summarised below.

Table 3.1: Syntax of Kind

K,L::=	Kind
*	Type
K -> L	Operators from K to L

Table 3.2: Syntax of Type

A,B::=	Type
X	Variable
Top	The biggest type at Kind *
A -> B	Function Type
$\{v_i\ l_i : B_i{}^{i \in 1....n}\}$	Record Type
A *extended by* B	Extended Record Type
$Forall\ X <: A :: K(B)$	Universal Type
$Object\ X\ A$	Object Type
$Class\ A$	Class Type
$Function\ X\ (B)$	Operator Type
$B[A]$	Operator Application Type

Table 3.3: Syntax of Program

a,b::=	Program
x	Variable
$\{l_i = b_i{}^{i \in 1....n}\}$	Record
a *extended by* b	Extended Record
$a.l$	Method Invocation
$a.l := b$	Field Update
$a.l\ := method\ (x : A)\ b\ end$	Method Update
$b[A]$	Constructor Application
$a[b]$	Application
$function\ (x : A)\ b\ end$	Function
$function\ (X <: A) b\ end$	Constructor Abstraction
$object\ (x : A)(a)$	Object Program
$Subclass\ (s1 : mu\ s2)\ s2 <: A$	
$of\ Program\ a : A\ with\ a\ override\ b$	Program Class

3.3 A Higher-Order Calculus

According to [15, 16] the definition of free variable and bound variable are as follows:

"A free variable is a notation that specifies places in an expression where substitution may take place".

"An occurrence of variable x is bound if it is in the body of a quantifier".

For an example the variable x becomes a bound variable, when we write, For all x, (x + 1)3 = x3 - 3x2 + 3x - 5 or there exists x such that x2 = 2.

The definition of substitution according to [17] is: "Substitution is a fundamental concept in logic. A substitution instance of a propositional formula is a second formula obtained by replacing symbols of the original formula by other formulas" [17].

3.3.1 Some Notations:

- A closed term is a term without free variable.
- We write $b\{x\}$ to highlight that x may occur free in b.
- We write $b\langle c\rangle$ instead of $b\langle x \leftarrow c\rangle$, i.e., substitute all free x with c in b.
- We identify programs that differ only by renaming bounded variables.
- We identify any two objects that differ only in the order of their components.

3.3.2 Structure of Rules

The calculus consists of a set of rules. Each rule has a number of premise judgments above a horizontal line and a single conclusion judgement below the line. Each judgement has the form $E \vdash \Im$ for a typing environment E and an assertion $\Im$. A premise of the form $E, E_{\mathrm{i}} \vdash \Im_{\mathrm{i}}$ forall $i \in 1...n$ is an abbreviation for n premises $E, E_1 \vdash \Im_1 ....E, E_{\mathrm{n}} \vdash \Im_{\mathrm{n}}$ if $n > 0$, and if $n = 0$ for $E \vdash \diamond$, which means that E is well-formed. Instead $j \in 1...n$ in the premise indicates that there are n separate rules, one for each j. Each rule has a name whose first word is determined by the conclusion judgement; for example, rule names of the form (*type*...) are for rules whose conclusion is a type judgement. So formation of a rule:

$$(Rule\ name) \qquad (Annotations)$$

$$\frac{E_1 \vdash \Im_1 \ldots\ldots\ldots\ldots\ldots\ldots E, E_{\mathrm{n}} \vdash \Im_{\mathrm{n}}}{E \vdash \Im} \tag{3.1}$$

3.3.3 Typing

The type rules of our language are formulated in terms of the following judgments:

Table 3.4: Judgments for Language

$E \vdash \diamond$	E is an environment
$E \vdash K\ kind$	K is a kind
$E \vdash A :: K$	Type A has kind K
$E \vdash B_i :: *$	Type B has kind * for i ∈ 1.....n
$E \vdash A \leftrightarrow B :: K$	A and B are equivalent type of kind K
$E \vdash A <: B :: K$	A is a sub type of B, both of kind K
$E \vdash a : A$	a is a value of type A
$E \vdash a \leftrightarrow b : A$	a is equal to b in type A.

Using these judgements and notations, we list the inference rules for our language. These rules are straightforward and listed below.

Kind Formation

$$(Kind\ *)$$

$$\frac{E \vdash \diamond}{E \vdash *\ kind} \tag{3.2}$$

For the environment E, if the conclusion assertion is a kind, then as a premise judgement we can state that, environment E is well-formed.

$$(Kind\ \text{->})$$

$$\frac{E \vdash K\ kind \qquad E \vdash L\ kind}{E \vdash K \text{ -> } L\ kind} \tag{3.3}$$

For environment E, if K and L are kinds, then for the same environment, as a conclusion judgement we can state that, K -> L is also a kind.

Type formation

$$(Type\ X)$$

$$\frac{E', X <: A :: K, E'' \vdash \diamond}{E', X <: A :: K, E'' \vdash X :: K} \tag{3.4}$$

According to the premise, we can conclude that X is a type in environment E″ having kind K. The following rule is straight forward. For any well-formed environment, the conclusion assertion can be any type having a valid kind.

$$(Type\ Top)$$

$$\frac{E \vdash \diamond}{E \vdash Top :: *} \tag{3.5}$$

$$(Type\ Record)(l_{\mathrm{i}}\ distinct, v_{\mathrm{i}} \in \{read/write, read, write\})$$

$$\frac{E \vdash B_{\mathrm{i}} \qquad \forall_{\mathrm{i}\,\in\,1....\mathrm{n}}}{E \vdash \{l_{\mathrm{i}} v_i : B_i^{\mathrm{i}\,\in\,1....\mathrm{n}}\}} \tag{3.6}$$

If the premise consists of some types, then for distinct level (l) and variance, the concluding assertion is a record type in the same environment E.

$$(Type\ Universal)$$

$$\frac{E, X <: A :: K \vdash B}{E \vdash Forall\ X <: A :: K(B)} \tag{3.7}$$

Within the premise the X is a sub type of A and has the kind K. This X is also a part of the environment, and the assertion of this judgement is a type B. So finally we can state that the assertion of the conclusion judgement is an universal type.

$$(Type\ Operator)$$

$$\frac{E, X :: K \vdash B :: L}{E \vdash \text{Function } X(B) :: K \text{ -> } L} \tag{3.8}$$

In the premise, if X is a part of the environment having a kind K and the assertion contains a type B having a kind L, we can state that the assertion of the conclusion judgement is an operator type having a kind K -> L.

$$\frac{E \vdash B :: K \text{ -> } L \qquad E \vdash A :: K}{E \vdash B[A] :: L} \quad (Type\ OpAppl) \tag{3.9}$$

If the premise consists of an operator type B having a kind K -> L and another type A having the kind K then the assertion of the conclusion judgement will be an operator application type of the kind L.

$$\frac{E, X \vdash A}{E \vdash \text{Object } X\ A} \quad (Type\ Object) \tag{3.10}$$

Type equivalence

$$\frac{E \vdash A \leftrightarrow B :: K}{E \vdash B \leftrightarrow A :: K} \quad (Type\ Eq\ Symm) \tag{3.11}$$

If A is equivalent to B, then we can say that B is equivalent to A.

$$\frac{E \vdash A \leftrightarrow B :: K \qquad E \vdash B \leftrightarrow C :: K}{E \vdash A \leftrightarrow C :: K} \quad (Type\ Eq\ Trans) \tag{3.12}$$

If A is equivalent to B and B is equivalent to C, then we can say that A is equivalent to C.

$$\frac{E \vdash X :: K}{E \vdash X \leftrightarrow X :: K} \quad (Type\ Eq\ X) \tag{3.13}$$

$$\frac{E \vdash \diamond}{E \vdash Top \leftrightarrow Top :: *} \quad (Type\ Eq\ Top) \tag{3.14}$$

$$(Type\ Eq\ Record)(l_{\mathrm{i}}, v_{\mathrm{i}} \in \{read/write, read, write\})$$

$$\frac{E \vdash B_i \leftrightarrow B_i\,' \quad \forall_i \in 1....n}{E \vdash \{l_i v_i : B_i{}^{\mathrm{i} \in 1....\mathrm{n}}\} \leftrightarrow \{l_i v_i : B_i{}'^{\,\mathrm{i} \in 1....\mathrm{n}}\}} \tag{3.15}$$

If B_i and B_i' are equivalent, then for distinct l and v the two record types will be equivalent.

$$\textit{(Type Eq Universal)}$$

$$\frac{E \vdash A \leftrightarrow A' :: K \quad E, X <: A :: K \vdash B \leftrightarrow B'}{E \vdash \text{Forall } X <: A :: K(B) \leftrightarrow \text{Forall } X <: A' :: K(B')} \tag{3.16}$$

For the premise above we can conclude that the two universal types are equivalent.

$$\textit{(Type Eq Operator)}$$

$$\frac{E, X :: K \vdash B \leftrightarrow B' :: L}{E \vdash \text{Function } X(B) \leftrightarrow \text{Function } X(B') :: K \text{ -> } L} \tag{3.17}$$

For the given premise, the two operator types in the conclusion are equivalent.

$$\textit{(Type Eq OPAppl)}$$

$$\frac{E \vdash B \leftrightarrow B' :: K \text{ -> } L \quad E \vdash A \leftrightarrow A' :: K}{E \vdash B[A] \leftrightarrow B'[A'] :: L} \tag{3.18}$$

For the given premise, the two operator application types in the conclusion are equivalent.

$$\textit{(Type Eval Beta)}$$

$$\frac{E, X :: K \vdash B\{X\} :: L \quad E \vdash A :: K}{E \vdash \text{Function } X(B)\{X\}[A] \leftrightarrow B\langle A\rangle :: L} \tag{3.19}$$

For this premise, in the conclusion, the operator application type is equivalent to substituting all the free variables X in B with A.

$$\textit{(Type Eq Object)}$$

$$\frac{E, X \vdash A \leftrightarrow A'}{E \vdash \text{Object } X\ A \leftrightarrow \text{Object } X\ A'} \tag{3.20}$$

Type inclusion

$$\textit{(Type Sub Refl)}$$

$$\frac{E \vdash A \leftrightarrow B :: K}{E \vdash A <: B :: K} \tag{3.21}$$

If A is equivalent to B then for the same kind K, A is a sub type of B.

$$(Type\ Sub\ Trans)$$

$$\frac{E \vdash A <: B :: K \qquad E \vdash B <: C :: K}{E \vdash A <: C :: K} \tag{3.22}$$

If A is a sub type of B and B is a sub type of C, then A is a sub type of C (if all have the same kind K).

$$(Type\ Sub\ X)$$

$$\frac{E', X <: A :: K, E'' \vdash \diamond}{E', X <: A :: K, E'' \vdash X <: A :: K} \tag{3.23}$$

$$(Type\ Sub\ Top)$$

$$\frac{E \vdash A :: *}{E \vdash A <: Top :: *} \tag{3.24}$$

For any valid type, Top is a super type. Equivalently, we can say that Top is the biggest Type.

$$(Type\ Sub\ Record)(l_i\ distinct)$$

$$\frac{E \vdash v_i B_i <: v_i{}' B_i{}' \quad \forall_i \in 1....n \qquad E \vdash B_i \qquad \forall_i \in n+1....n+m}{E \vdash \{l_i v_i : B_i{}^{\,i \in 1....n+m}\} <: \{l_i v_i : B_i{}'^{\,i \in 1....n}\}} \tag{3.25}$$

A record type is a sub type of other record types, if it has all the components of the other record type plus some more.

$$(Type\ Sub\ Universal)$$

$$\frac{E \vdash A <: A' :: K \quad E, X <: A :: K \vdash B <: B'}{E \vdash \text{Forall } X <: A :: K(B) <: \text{Forall } X <: A' :: K(B')} \tag{3.26}$$

If X is a sub type of A with kind K, A is a sub type of A′ with kind K, and B is a sub type of B′, then in the conclusion judgement the left universal type is a sub type of the right universal type.

$$(Type\ Sub\ Operator)$$

$$\frac{E, X :: K \vdash B <: B' :: L}{E \vdash \text{Function } X\ (B) <: \text{Function } X\ (B') :: K \text{ -> } L} \tag{3.27}$$

In the conclusion judgement, Function X (B) is a sub type of the other function type, if B is a sub type of B′ and X is a part of the environment having kind K.

$$\textit{(Type Sub OPAppl)}$$
$$\frac{E \vdash B <: B' :: K \text{ -> } L \quad E \vdash A :: K}{E \vdash B[A] <: B'[A] :: L} \tag{3.28}$$

If B is a sub type of B′ each having kind K -> L and A is a type having kind K then we can say that operator application type B[A] is a sub type of the other operator application type each having kind L.

$$\textit{(Type Sub Object)}$$
$$\frac{E \vdash \text{Object } X\ A \quad E \vdash \text{Object } Y\ B \quad E, Y, X <: Y \vdash A <: B}{E \vdash \text{Object } X\ A <: \text{Object } Y\ B} \tag{3.29}$$

If X is a sub type of Y and A is a sub type of B, then for two types "Object X A" and "Object Y B", we can conclude that the first is a sub type of the second.

$$\textit{(Type Sub Invariant)}$$
$$\frac{E \vdash B}{E \vdash^{read/write} B <:^{read/write} B} \tag{3.30}$$
$$\textit{(Type Sub Covariant)}$$
$$\frac{E \vdash B <: B' \qquad v \in \{^{read/write, read}\}}{E \vdash vB <:\ ^{read}B'} \tag{3.31}$$
$$\textit{(Type Sub Contravariant)}$$
$$\frac{E \vdash B' <: B \qquad v \in \{^{read/write, write}\}}{E \vdash vB <:\ ^{write}B'} \tag{3.32}$$

The above three rules are trivial. Here read means a component of a tuple which is covariant, write means the input of a function and read/write means the invariant component. For further reading review Section 2.3.

Program typing

$$\textit{(Program Subsumption)}$$
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B} \tag{3.33}$$

If a is a program of type A and A is a sub type of B then the program a also has the type B.

$$(Program\ x)$$

$$\frac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x : A} \tag{3.34}$$

If x is a program of type A and it is also a part of an environment, then it is certain that the type of program x is A.

$$(Program\ Record)$$

$$\frac{E, x_i : A \vdash b_i : B_i \qquad \forall_i \in 1....n \qquad E \vdash A \leftrightarrow \{l_i v_i : B_i\ ^{i \in 1....n}\}}{E \vdash \{l_i = b_i\ ^{i \in 1....n}\} : A} \tag{3.35}$$

If A is a record type and program b_i is the type of B_i (for i = 1 to n) then the conclusion judgement is a record program having type A.

$$(Program\ MetInvocation)$$

$$\frac{E \vdash a : \{l_i v_i : B_i\ ^{i \in 1....n}\} \qquad v_j \in \{read/write, read\} \qquad j \in 1.....n}{E \vdash a.l_j : B_j} \tag{3.36}$$

If a is a record program and we want to select a component from a, the type of the conclusion judgement will be the type of the selected element.

$$(Program\ MetUpdate)\ (where\ A \equiv \{l_i v_i : B_i\ ^{i \in 1....n}\})$$

$$\frac{E \vdash C <: A \qquad E \vdash a : C \qquad E, x : C \vdash b : B_j \quad v_j \in \{read/write, write\} \qquad j \in 1.....n}{E \vdash a.l_j := \text{method } (x : C)\ b \text{ end} : C} \tag{3.37}$$

$$(Program\ FieldUpdate)\ \ (where\ A \equiv \{l_i v_i : B_i\ ^{i \in 1....n}\})$$

$$\frac{E \vdash B_j <: C \qquad E \vdash a : A \quad v_j \in \{read/write, write\} \qquad j \in 1.....n}{E \vdash a.l_j := b : C} \tag{3.38}$$

From the above two rules, we can state that if we update any components of a program, the type of the updated program will be the same type as before or a sub type of the previous type.

$$(Program\ ConsAbstraction)$$

$$\frac{E, X <: A :: K \vdash b : B}{E \vdash \text{function}(X <: A :: K)\ b \text{ end} : \text{Forall } X <: A :: K\ (B)} \tag{3.39}$$

$$(Program\ ConsApplication)$$
$$\frac{E \vdash b : \text{Forall } X <: A :: K\ (B)\{X\} \qquad E \vdash A' <: A :: K}{E \vdash b[A'] : B\,\langle A' \rangle} \tag{3.40}$$

The above two rules are trivial. For the premise judgements it is clear that the conclusion judgements are constant abstraction and constant application program sequentially.

3.4 Proof Rules

Depending on typing rules, a list of proof rules has been implemented which makes the calculus strong. They are stated below. Here a == b :A means that a and b are equal program having the same type A.

3.4.1 Basic Rules

$$(Proof\ Symmetry)$$
$$\frac{E \vdash a == b : A}{E \vdash b == a : A} \tag{3.41}$$

If a is equivalent to b each having type A, we can say that b is equivalent to a.

$$(Proof\ Transitivity)$$
$$\frac{E \vdash a == b : A \qquad E \vdash b == c : A}{E \vdash a == c : A} \tag{3.42}$$

If a is equivalent to b and b is equivalent to c, we can say that a is equivalent to c.

$$(Proof\ Subsumption)$$
$$\frac{E \vdash a == b : A \qquad E \vdash A <: B}{E \vdash a == b : B} \tag{3.43}$$

If a is equivalent to b each having type A, and A is a sub type of B, then we can say that a is equivalent to b each having type B.

$$(Proof\ Name)$$
$$\frac{E \vdash a\{P\} == b : A}{E \vdash a\{Name\} == b : A} \tag{3.44}$$

3.4.2 Function Rules

$$\frac{\begin{array}{c}(Proof\ Fun\ Declaration)\\ E, x : A \vdash a == b : B\end{array}}{E \vdash function(x : A)\ a\ end == function(x : A)\ b\ end : A\ \text{->}\ B} \tag{3.45}$$

If a is equivalent to b, and each has the type B, and program x has the type A, we can say that these two functions are equivalent.

$$(Proof\ Fun\ Application)$$

$$\frac{E \vdash a1 == a2 : A\ \text{->}\ B \qquad E \vdash b1 == b2 : A}{E \vdash a1[b1] == a2[b2] : B} \tag{3.46}$$

If a1 and a2 are two equivalent function Programs and b1 and b2 are of the type A, we can say that a1[b1] is equivalent to a2[b2], both having type B.

$$(Proof\ Fun\ Beta)$$

$$\frac{E \vdash \diamond}{E \vdash function(x : A)\ a\ end[b] == a\,\langle b\rangle : B} \tag{3.47}$$

This proof rule is distinctive and very important. For any well-formed environment we can say that applying program b to a function, is equivalent to substituting all other free occurrences of x in this function program with b.

3.4.3 Record Rules

$$\frac{\begin{array}{c}(Proof\ Rec\ Declaration)\\ E \vdash a_i == b_i : A_i \quad i = 1....n\end{array}}{E \vdash \{l_i = a_i\}^{i=1....n...m1} == \{l_i = b_i\}^{i=1....n...m2} : \{l_i : A_i\}^{i=1..n}} \tag{3.48}$$

Two record programs are equivalent if they have the same type, and any common or equivalent components.

$$(Proof\ Rec\ Selection)$$

$$\frac{E \vdash a == b : \{l_i : A_i\}}{E \vdash a.l_i == b.l_i : A_i} \tag{3.49}$$

If a and b are two equivalent records, then selecting a record element from a, would be equivalent to selecting the same component from b.

$$(Proof\ Rec\ Sel\ Beta)$$

$$\frac{E \vdash \diamond}{E \vdash \{l_i = a_i\}.l_i == a_i : A_i} \tag{3.50}$$

This proof rule is also quite important. For any well-formed environment, we can say that record element selection is equivalent to the value of that element.

$$(Proof\ Rec\ Update)$$

$$\frac{E \vdash a == b : \{l_i : A_i\} \qquad E \vdash a' == b' : A_i}{E \vdash a.l_i := a' == b.l_i := b' : \{l_i : A_i\}} \tag{3.51}$$

If we update the same component from two equivalent records by two equivalent programs sequentially, the updated records will be equivalent.

$$(Proof\ Rec\ Upd\ Beta)$$

$$\frac{E \vdash \diamond}{E \vdash \{l_i : a_i\}^{i=1 \ldots n}.l_i := b == \{l_i : a_i\}^{i=1 \ldots n}\,\langle b/a\rangle : \{l_i : A_i\}^{i=1 \ldots n}} \tag{3.52}$$

This proof rule is important because it is frequently the last rule used in a proof. This is due to the simplicity of the premise. For any well-formed environment, we can say that the record update is equivalent to substituting the value of that element, with the new value.

3.4.4 Extended Record Rules

$$(Proof\ ExtRec\ Declaration)$$

$$\frac{E \vdash a1 == b1 : \{l_i : A_i\}^{i=1 \ldots n} \qquad E \vdash a2 == b2 : \{l_i : A_i\}^{i=n+1 \ldots m}}{E \vdash a1\ extended\ by\ a2 == b1\ extended\ by\ b2 : \{l_i : A_i\}^{i=1 \ldots n \ldots m}} \tag{3.53}$$

In the premise, if we have four records where the first two are equivalent and the second two are equivalent (call each a pair), then the extended record formed by the first of each of the two pairs is equivalent to the extended record formed by the second two in each pair.

$$\frac{E \vdash \diamond}{E \vdash \{l_i = ai\}^{i=1...n} \text{ extended by } \{l_i = a_i\}^{i=n+1...m} == \{l_i : a_i\}^{i=1...n...m}} \quad (Proof\ ExtRec\ Beta) \tag{3.54}$$

If we divide a record into two parts, then the extended record formed by this two parts is equivalent to the whole record.

3.4.5 Object Rules

(*Proof Object Declaration*)

$$\frac{E \vdash a.l_i == b.l_i : A_i \langle Object(Self)\{l_i : A_i\}/Self\rangle \quad i = 1....n}{E \vdash a == b : Object(Self)\{l_i : A_i\}} \tag{3.55}$$

If we have two equivalent object selection programs, then we can say that two object programs are also equivalent.

(*Proof Obj Selection*)

$$\frac{E \vdash a == b : object(self : A)\{l_i : A_i\}}{E \vdash a.l_i == b.l_i : A_i} \tag{3.56}$$

If we have two equivalent object programs, then selecting the same component from each of them will be equivalent too.

(*Proof Object Sel Beta*)

$$\frac{E \vdash \diamond}{E \vdash object(self : A)\{l_i = a_i\}.l_i == a_i \langle object(self : A)\{l_i = a_i\}/self\rangle} \tag{3.57}$$

This proof rule is distinctive and very important. For any well-formed environment, in the conclusion judgement the method invocation program is equivalent to substituting all the free occurrences of `self` into the value of the element, with object program.

(*Proof Obj Update*)

$$\frac{E \vdash a == b : object(self : A)\{l_i : A_i\} \qquad E \vdash a' == b' : A_i}{E \vdash a.l_i := a' == b.l_i := b' : object(self : A)\{l_i : A_i\}} \tag{3.58}$$

If we update the same component from two equivalent objects by two equivalent programs sequentially, the updated objects will be equivalent.

$$\frac{(Proof\ Object\ Upd\ Beta)\qquad E \vdash \diamond \quad i = 1....n}{E \vdash object(self : A)\{l_i = a_i\}.l_i := method(x : A)b\ end == object(self : A)\{l_i = b\,\langle self/x \rangle\}} \tag{3.59}$$

This proof rule is distinctive and very important. For any well-formed environment, in the conclusion judgement the method update program is equivalent to substituting all the free occurrences of x in the value of the element, with `self`.

3.4.6 Polymorphic Rules

$$(Proof\ Poly\ Declaration)$$

$$\frac{E, X <: A \vdash a == b : B}{E \vdash function(X <: A)a\ end == function(X <: A)b\ end : Forall\ (X <: A)B} \tag{3.60}$$

In the premise judgement if we have two equivalent programs a and b, and X is a sub type of A, in the conclusion judgement the two constructor abstraction programs will be equivalent.

$$(Proof\ Poly\ Application)$$

$$\frac{E \vdash a1 == b1 : Forall\ (X <: A)B \qquad E \vdash A1 \leftrightarrow A2}{E \vdash a1[A1] == b1[A2] : B\,\langle A1/X \rangle} \tag{3.61}$$

If we have two constructor abstraction programs and we apply two equivalent types to them sequentially, resultant constructor application programs will be equivalent.

$$(Proof\ Poly\ Beta)$$

$$\frac{E \vdash \diamond}{E \vdash function(X <: A)a\ endend[B] == a\,\langle B/X \rangle : C\,\langle B/X \rangle} \tag{3.62}$$

This proof rule is distinctive and very important. For any well-formed environment, in the conclusion judgement the constructor application program is equivalent to substituting all the free occurrences of X in a, with B.

Chapter 4

System

In this chapter we explain the system in detail. This includes the programming language Haskell, the implementation of our language, the implementation of the toolkit, as well as a user manual.

4.1 Haskell

The language we use to develop our system is Haskell [10]. It is a functional programming language. Haskell is strongly typed but its type system is much less restrictive because it supports polymorphism. Lazy evaluation is another of its powerful features, i.e., it will evaluate a program only if its value is required. Besides polymorphism higher order functions is the main abstraction mechanism available in Haskell, i.e., the language allows functions to be parameters as well as return values of other functions. Furthermore the code is easy to understand, re-usable, and easy to maintain. Its focus is on what is to be computed, not how it should be computed.

There are a variety of implementations available for Haskell. The current Haskell platform is Haskell Platform 2011.2. In our system we use Haskell Platform 2010 and GHC 6.10.3, but it will also run in the current platform. Everything is freely available for Windows, Mac and Linux at http://hackage.haskell.org/platform/ [10].

4.2 Parsec

Parsec is an industrial strength, monadic parser combinatory library for Haskell. It can parse context-sensitive, infinite look-ahead grammars, but it performs best on predictive grammars. It is simple, fast, safe, well documented, has extensive libraries and user friendly error messages. It is distributed with an unrestrictive BSD style license. The most general way to run a parser is to use the `runParser` function. `runParser p st filePath input`, runs parser `p` on the input list of tokens `input`, obtained from source `filePath` with the initial user state `st`. The filePath is only used in error messages and may be the empty string. It returns either a ParseError (Left) or a value of type a (Right) [14].

4.3 Language Implementation

The implementation has two steps. The first phase contains the implementation of the language itself as well as its calculus. The second phase is the toolkit to verify object oriented programs written in our language. The first phase also contains a couple of parsers, (i.e.,kind parser, type parser, program parser, type declare parser, program declare parser) and different supporting functions. We explain the parsers and a couple of important functions in detail in the following sub sections.

4.3.1 Kind Parser

According to the syntax of `Kind` a kind parser has been implemented. `Kinds` are represented in the Haskell program by elements of the data type `Kind`. The type definition of this `Kind` is as follows:

```
infixr 1 :>>
data Kind = Ty | Kind :>> Kind
```

For this data type `Kind`, we derived an instance declaration of the class Eq, i.e., a class giving equality and inequality, as well as an instance of show and read also defined for `Kind`. The string representation of `Kind` in our language is represented by * . The Haskell element into which the string is translated by the parser is `Ty`. Some sample examples of kind parser are stated below.

Table 4.1: Sample Kind Example

Input
$Ty :>> Ty :>> Ty :>> Ty$
$(((* \text{->} *) \text{->} (* \text{->} *)) \text{->} *) \text{->} *$
$* \text{->} * \text{->} *$
$(Ty :>> Ty) :>> (Ty :>> Ty)$

4.3.2 Variance Parser

There is a variance parser that works inside the type parser. The type definition of `Variance` is as follows:

```
data Variance = R | W | RW
```

For `Variance` data type we derived an instance declaration of the class Eq, i.e., a class giving equality and inequality, as well as an instance of show and read also defined for this data type. Some examples about variance parser are stated below.

Table 4.2: Sample `Variance` Representation

Haskell Element	String Representation
R	*read*
W	*write*
RW	*read/write*

4.3.3 Type Parser

The type definition of the data type `Type` is as follows:

```
data Type = TypeVar String
  | TypeName String
  | Top
  | Type :-> Type
  | RecordType (M.Map String (Variance,Type))
  | ExtRecType Type Type
  | ObjectType String Type
  | ClassType String Type
  | UniversalType String Type Kind Type
  | Operator String Kind Type
  | OpAppl Type Type
```

Before we go into the details, we want to explain shortly the alternative for a couple of non trivial data types. The `TypeVar` alternative is variable type, i.e., any capital letter or any string begins with capital letter. The `Top` is the biggest type, the internal representation of a record is represented by the `RecordType`. The alternative for `Operator` is function and `OpAppl` represents the operator or function application.

We defined an instance of the show class, i.e., a class that convert a value to a string, for the data type `Type`. The type definition of `TypeVarEnv` is

```
type TypeVarEnv = M.Map String (Type,Kind)
```

The result of this parser is the tuple (Type,Kind). TypeVarEnv works as an internal state of the parser by saving any defined type variable with their Type and Kind. The examples below show some input and output for the typeParser.

Example Top

```
Input: "Top"
Output: (Top,Ty)
```

Example RecordType

```
Input: "{l:Top, read m:Top}"
Output: (RecordType (fromList [("l",(RW,Top)),("m",
(R,Top))]), Ty)
```

The input is a RecordType. Here as a Variance one can give as input: read, write or nothing at all. If read is the input, the output will show R, for write the output will show W and for no input the default output will be RW.

Example ObjectType

```
Input: "Object  X {m:Top,l:Top->Top}"
Output: (ObjectType X (RecordType (fromList [("m",(RW,Top)),
("l",(RW,Top :-> Top))]),Ty)
```

In this ObjectType input, X is a TypeIdentifier, i.e., upper case character or any string start with capital letter. the default TypeIdentifier for this Type is Self). The body of a ObjectType needs to be the RecordType.

Example ClassType

```
Input: "Class {m :Self,l:Top->Top}"
Output: (ClassType Self (RecordType (fromList [("m",(RW,Self)),
        ("l",(RW,Top :-> Top))]),Ty)
```

This is a ClassType input. After the basic token Class there is no TypeIdentifier, that means the TypeIdentifier Self will work as a default input. For this Type the body always needs to be the RecordType.

Example UniversalType

```
Input: "Forall X<:Top::* (Top->Top)"
Output: (UniversalType X Top Ty (Top->Top),Ty)
```

For this `UniversalType`, `X` needs to be a sub type of the `Type Top` and they also need to have the same `Kind`.

Example Operator

```
Input: "Function X (Top)"
Output: (Operator X Ty Top,Ty->Ty)
```

In this `OperatorType Function` is the basic token, `X` is a `TypeIdentifier` that works as a parameter of a function. `Top` is the return type of that function. The `Kind` is not specified, so `Ty` will work as a default `Kind` input.

Example OpAppl

```
Input: "Function X (Top) [Top]"
Output: (OpAppl (Operator X Ty (Top))[Top],Ty)
```

The first part of this `OpAppl` needs to be an `Operator Type`. The second part can be any `Type`.

Example ExtRecType

```
Input: "{read m :Top} extended by {read o :Object Self {p:Self}}"
Output:Right (ExtRecType (RecordType (fromList [("m",(R,Top))]))
      (RecordType (fromList [("o",(R,ObjectType Self (RecordType
      (fromList [("p",(RW,Self))])))))])),Ty)
```

This `ExtRecType` is actually two or more `RecordType` separated by the basic token `extended by`.

Example Type :-> Type

```
Input: "{ }->{ }"
Output: (RecordType (fromList [])) :->(RecordType
      (fromList [])), Ty)
```

This is straight forward and applicable for any `Type`.

4.3.4 Some Important Functions

There are many functions in the type module. A few are necessary for using the system and are really important. As mentioned above, for `Type` we already defined an instance of the show class. So for all functions, show is applied to their result automatically. A couple of functions are explained below.

typeNormalForm

The type declaration of `typeNormalForm` is stated below with the type declaration of `TypeNameEnv`.

```
type TypeNameEnv = M.Map String (Type,Kind)
typeNormalForm :: TypeNameEnv -> Type -> Type
```

This function takes a `Type` as a parameter and returns a `Type` in a normal form. A couple of examples are stated below.

Example 1

```
Input: typeNormalForm M.empty (TypeVar "Y")
Output: Y
```

Example 2

```
Input: typeNormalForm (M.empty) (RecordType (M.fromAscList
       [("l",(R,Top))]))
Output: {read l : Top}
```

Example 3

```
Input: typeNormalForm (M.empty) (OpAppl (Operator "X" Ty
      (TypeVar "X" :-> TypeVar "X")) Top)
Output: Top->Top
```

typeSubstitute

```
typeSubstitute :: Type -> String -> Type -> Type
```

The `typeSubstitute` function takes three parameters, namely `Type`(t1), `String`(s) and `Type`(t2). As an output it returns a `Type` after substituting all the free occurrences of s at t2 with t1. A couple of input and output for this function are stated below.

Example 1

```
Input: typeSubstitute (TypeVar "X") "Y" (TypeVar "Z")
Output: Z
```

Example 2

```
Input: typeSubstitute (TypeVar "X") "Y" (TypeVar "Y")
Output: X
```

Example 3

```
Input: typeSubstitute (TypeVar "X") "Y" (RecordType
(M.fromAscList [("l",(R,(TypeVar "Y")))]))
Output: {read l : X}
```

freeTypeVars

```
freeTypeVars :: Type -> S.Set String
```

The `freeTypeVars` function takes one parameter, namely `Type(t1)` and returns a set of free `String` in t1. A couple of input and output for `freeTypeVars` function are stated below.

Example 1

```
Input: freeTypeVars (TypeVar "X")
Output: fromList ["X"]
```

Example 2

```
Input: freeTypeVars (ObjectType "X" (RecordType
(M.fromAscList [("l",(R,(TypeVar "X")))])))
Output: fromList []
```

Example 3

```
Input: freeTypeVars(Class "Self"(Operator "X" Ty (RecordType(M.
fromAscList[("l",(R,TypeVar"Y"))])))(Operator "X" Ty(RecordType
M.empty))(RecordType(M.fromAscList [("l",(R,(TypeVar "X")))])))
Output: fromList ["X","Y"]
```

typeEqual

```
typeEqual :: TypeNameEnv -> Type -> Type -> Bool
typeEqual nenv t1 t2 = typeEqualNF (typeNormalForm nenv t1)
                       (typeNormalForm nenv t2)
typeEqualNF :: Type -> Type -> Bool
```

As parameter the `typeEqual` function takes `TypeNameEnv`, `Type`(t1) and `Type`(t2). As an output it returns `Bool` type by applying `typeEqualNF` to these two `Type` t1 and t2. Three examples below show the input and output for the function `typeEqualNF`.

Example 1

```
Input:  typeEqual (M.empty) (TypeVar "X") (TypeVar "Y")
Output: False
```

Example 2

```
Input: typeEqual (M.empty) (TypeVar "X") (TypeVar "X")
Output: True
```

Example 3

```
Input: typeEqual (Operator "X" Ty Top) (Operator "Y" Ty Top)
Output: True
```

In Example 3, though the parameter of the two Operator types is different, they are equal types. The implementation of `typeEqualNF` for `Operator Type` is stated below.

```
typeEqualNF (Operator s1 k1 t1) (Operator s2 k2 t2) =
   let fv = freeTypeVars t1 `S.union` freeTypeVars t2
       sn = TypeVar (newTypeVar "X" fv)
   in typeEqualNF (typeSubstitute sn s1 t1) (typeSubstitute sn s2 t2)
```

subType

```
subType :: TypeEnv -> Type -> Type -> Bool
subType (venv,nenv) t1 t2 = subTypeNF venv (typeNormalForm nenv t1)
                            (typeNormalForm nenv t2)
```

```
subTypeNF :: TypeVarEnv -> Type -> Type -> Bool
lubType :: TypeVarEnv -> Type -> Maybe Type
```

The subType function determines if one Type is sub type of other Type or not. As a parameter it takes TypeEnv, Type(sub type) and Type(super type). Its return type is Bool. The two assisting functions are subTypeNF and lubType. subTypeNF takes TypeVarEnv and two normal formed Type (sub type and super type) and returns type Bool. A couple of examples of subType are stated below.

Example 1

```
Input:  subType (M.empty,M.empty) (TypeVar "X") Top
Output: True
```

As we stated Top is super type for all other Type, so whatever the sub type is, if Top is a super type then it is always true.

Example 2

```
Input: subType (M.empty,M.empty) (RecordType (M.fromAscList
   [("l",(R,TypeVar"Y")),("m",(RW,Top))])) (RecordType (M.fromAscList
   [("l",(R,TypeVar"Y"))]))
Output: True
```

Example 3

```
Input: subType (M.empty,M.empty) (Operator "X" Ty Top) (Operator
       "Y" Ty Top)
Output: True
```

4.3.5 Type Declaration Parser

In order to declare a Type we will have to start with the basic token type followed by one to many TypeIdentifier. After that, we require the basic token equal (=) and a valid Type to complete the type declaration. A couple of input and output examples are stated below.

Example 1

```
Input:  runTDecl "type X = {}"
Output: fromList[("X",({},*))]
```

Example 2

```
Input: runTDecl "type X Y= {}"
Output: fromList[("X",(Function Y::*({}),* -> *))]
```

Example 3

```
Input: runTDecl "type X Y Z= {}"
Output: fromList[("X",(Function Y::*(Function Z::*({})),* -> *->*))]
```

4.3.6 Program Parser

The type definition of the data type Program is as follows:

```
data Program = ProgVar String
     | ProgName String
     | Function String Type Program
     | Appl Program Program
     | Record (M.Map String Program)
     | ExtRecord Program Program
     | ProgClass String String Type Program Type Program Program
     | MetInvocation Program String
     | FieldUpdate Program String Program
     | MetUpdate Program String String Type Program
     | Object String Type Program
     | ConsAbstraction String Type Kind Program
     | ConsApplication Program Type
```

Here ProgVar stands for program variable, i.e, any small letter or any string starts with a small letter. The Appl represents the application of a Program to other Program. In order to select an element from a Record or Object we use MetInvocation. By using MetUpdate we can update a method inside an Object. The alternative for ConsAbstraction and ConsApplication are constant abstraction and constant application respectively.

For this data type Program, we defined an instance of show class, i.e., a class which converts a value to a String. Examples below show some input and output for the program parser.

Example Function

```
Input: function (x:Object {}) {} end
Output: (Function x (ObjectType "Self" (Record
(M.fromAscList []))) (Record (M.fromAscList [])),
(ObjectType "Self" (Record (M.fromAscList []))) ->
 (RecordType (M.fromAscList [])))
```

`Function Program` always begins with the basic token `function` followed by a opening bracket and a `ProgramIdentifier`, i.e., lower case character or any string start with small letter. After that we need to put the basic token `colon`(:), the `Type` of that `ProgramIdentifier`, and the closing bracket. As a last step we need to specify the `Program` (body of the function) and the final basic token `end`.

Example ConsAbstraction

```
Input: function (X<:Object {}) {} end
Output: (ConsAbstraction X (ObjectType "Self" (Record
(M.fromAscList []))) TY (Record (M.fromAscList []))),Forall X
(ObjectType "Self" (Record (M.fromAscList []))) TY
(Record (M.fromAscList []))))
```

For constant abstraction `Program (ConsAbstraction)` the starting basic token is `function`. Then we need to put opening and closing first bracket. Inside this bracket we need to specify the `TypeIdentifier` followed by basic token `subtype` and the `Type` of the `TypeIdentifier`. After that we will have to specify the body `Program` and the last basic token `end`.

Example Record

```
Input: {l={}}
Output: ((Record (M.fromAscList [("l",(Record (M.fromAscList []
  )))])),(RecordType (M.fromAscList [("l",(RW,(RecordType
  (M.fromAscList [])))))])))
```

`Record Program` starts with the left second bracket followed by zero to many record elements separated by comma and the right second bracket. Each record element is the combination of a `ProgramIdentifier` followed by basic token `equal` (=) and a `Program`.

Example Object

```
Input: object (x:Object {l=Self})({m=x})
Output: (Object x Object Self (RecordType (M.fromAscList [("l",
 (RW,Self))])) (Record (M.fromAscList [("l",x)],Object Self
 (RecordType (M.fromAscList [("l",(RW,Self))])))
```

`Object Program` is straight forward. The `Type` of the `ProgramIdentifier` needs to be `ObjecType` and the body of this `Object Program` needs to be a `Record Program`.

Example ProgClass

```
Input: Subclass (self:mu X) where X <:Function Y ({l:Top})
       with {} override {}
Output: (ProgClass self X Operator Y Ty (RecordType
 (M.fromAscList [("l",(RW,Top))])) (Object "self" (ClassType
 "Self" (RecordType M.empty) ) (Record (M.fromAscList [("new",
 (Object "self" (ObjectType "Self" (RecordType M.empty))(Record
 M.empty)))]))) (ClassType "Self" (RecordType M.empty)) (Record
 (M.fromAscList [])) (Record (M.fromAscList [])), ClassType "Self"
 (RecordType M.empty))
```

In `ProgClass Program`, two `Typeidentifier(X)` need to be equal and sub type of the `Operator Type` whose body is a `Type` of `RecordType`. It can be followed by the optional `superClasses`. One sample `superClasses` is as follows: `of Program (p) (Object) : Type (t) (ClassType)`. The return type of `Program (p)` needs to be the sub type of the given `Type (t)`. The `Operator Type` generated form this `superClasses` needs to be super type of the `Operator Type` in `Subclass` . If no `superClasses` is provided then by default it will return an `Object Program` having a `Class Type` with empty `RecordType`, and a Record Program with single element `new` of `program self`. Now we need to add the basic token `with` and a `Record Program` as well as basic token `override` and a `Record Program`.

Example Appl

```
Input: function (x:Top) {} end [{}]
Output: ( Appl (Function x Top (Record (M.fromAscList [])))
       (Record (M.fromAscList [])), (RecordType (M.fromAscList [])))
```

The Appl Program takes two parameters. The first Program needs to be a Function Program. The Type of the second Program needs to be a sub type of the Type of the ProgramIdentifier in Function Program.

Example ConsApplication

```
Input: function (X<:Object {}) {} end [Object {}]
Output: (ConsApplication (ConsAbstraction X (ObjectType "Self"
(Record (M.fromAscList []))) TY (Record (M.fromAscList [])))
(ObjectType  "Self" (Record (M.fromAscList []))),(RecordType
(M.fromAscList [])))
```

ConsApplication takes a ConsAbstraction Program followed by a Type (t). The super type of the TypeIdentifier needs to be sub type of the given Type (t).

Example MetInvocation

```
Input: {l={}}.l
Output: (MetInvocation (Record (M.fromAscList [("l",(Record
 (M.fromAscList [])))])) l,(RecordType (M.fromAscList [])))
```

The syntax of the method invocation (MetInvocation) is a Program (p) followed by the basic token select (.) and a ProgramIdentifier. The Program (p) can be a Record Program or Object Program(body is a Record Program). The ProgramIdentifier needs to be a member of Record Program.

Example FieldUpdate

```
Input: {l={}}.l:= {m={}}
Output: (FieldUpdate (Record (M.fromAscList [("l",(Record
 (M.fromAscList [])))])) l (Record (M.fromAscList [("m",(Record
 (M.fromAscList [])))])),(RecordType (M.fromAscList [("l",(RW,(Record
 (M.fromAscList [])))]))))])))
```

Program FieldUpdate starts with a Record Program followed by basic token select (.), ProgramIdentifier, basictoken update (:=) and another Program (p2). The Type of the ProgramIdentifier needs to be a super type of the Type of Program (p2).

Example MetUpdate

```
Input: object (x:Object {l:{}}) ({l={}}).l:=method(l: Object {l:{}}))
{m={}} end
Output: (MetUpdate (Object (x:ObjectType "Self"(RecordType (M.
fromAscList [("l",(RW,(RecordType (M.fromAscList [ ]))))]))) ((Record
(M.fromAscList [("l",(Record (M.fromAscList []))))])))) l l(ObjectType
"Self" (RecordType (M.fromAscList [("l",(RW,(RecordType (M.fromAscList
[ ]))))]))) ((Record (M.fromAscList [("m",(Record(M.fromAscList []))))]
))),ObjectType (RecordType (M.fromAscList [("l",(RW,(RecordType
(M.fromAscList [ ]))))])))
```

Method update `MetUpdate` starts with an `Object Program (p1)` followed by basic token `select (.)`, `ProgramIdentifier (l1)`, basictoken `update (:=)`, basic token `method`, `ProgramIdentifier`, `Type (t1)`,a `Program (p2)` and final basic token `end`. The `Type` of `Program p1` needs to be `subType` of the `Type t1` as well as `Type` of `Program p2` also need to be sub type of the functions `(typeSubstitute t1 s t)` `Type` ,i.e., substitute all free `s` in `t` with `t1`, where `t1` is the `Type` of the `Program p1`, `s` is the `ProgramIdentifier` inside the `Program p1` (for above example it is `x`) and `t` is the `Type` of the label `l1`.

4.3.7 Program Declare Parser

```
Input: x : {l:{}} = {l={}}
Output: ()
```

This is one sample input and output for program declare parser. It takes a `ProgramIdentifier` followed by basic token `colon (:)`, a `Type` (t1), basic token `equal (=)` and a `Program`. The `Type` of this `Program` needs to be sub type of `Type` t1.

4.4 Calculus Implementation

The calculus consists a set of proof rules. The type defination of `ProofRules` is as follows:

```
data ProofRules = Symm
                 | Name
                 | Tran Program Type
                 | Sub  Type
                 | Func
                 | Beta
                 | FuncAppl Type
                 | ExtRec Type Type
                 | ExtNormal
                 | RecEq
                 | MetInv4Rec Type
                 | MetInvEq4Rec
                 | MetInv4Obj Type
                 | MetInvEq4Obj
                 | FieldUp
                 | FieldUp4Obj
                 | FieldUpEq
                 | GetMetInv4Obj String
                 | MetUpdateEq
                 | ConsAbs
                 | ConsAppl String Kind
                 | ConsApplEq
```

The type `ProofRules` has made a member or instance of `Show` class by defining the signature functions for this type. Related to `ProofRules` a type `UndoProofRules` has been defined. Tye type defination for `UndoProofRules` is as follows:

```
data UndoProofRules = USymm
     | UName Program
     | UTran
     | USub  Type
     | UFunc String Type
     | UBeta String Type Program Program Type
```

```
| UFuncAppl
| UExtRec
| UExtNormal Program Program Type
| URecEq Program Program Type
| UMetInv4Rec String
| UMetInvEq4Rec Program String Program Type
| UMetInv4Obj String
| UMetInvEq4Obj String Type Program String Type
| UFieldUp String
| UFieldUp4Obj String
| UFieldUpEq Program String Program Type
| UGetMetInv4Obj String Type
| UMetUpdateEq String Type Program String String Type Program Type
| UConsAbs String Type Kind
| UConsAppl Type Type
| UConsApplEq String Type Kind Program Type Type
```

In order to apply the `Proof Rules` stated in chapter two, we need a `Theorem` and `ProofState`. Type definition of this two types are as follows:

```
newtype Theorem = Th (Program, Program,Type)
type ProofObligations = [Theorem]
type ProofState = (ProgramVarEnv,TypeVarEnv,ProofObligations)
```

4.4.1 Parser

Two simple parsers have been implemented to assist user for giving correct input. This two parsers are explained below with a couple of input and output samples.

proofParser

The `proofParser`, parse the first `Theorem`. It takes a `Program` follwed by two `equal` basic token (==), another `Program`, basic token colon(:) and finally the `Type` of this two `Program`. It returns `Theorem` or user friendly error message. a couple of sample input and outpur for this parser are as follows:

```
Input: "{}=={}:{}"
Output: {}=={}:{}
```

```
Input: "true==true:Bool"
Output: true==true:Bool
```

```
Input: "(true.or)[false]==true:Bool"
Output: true.or[false]==true:Bool
```

doRuleParser

In order to proof properties of a program, sometimes user needs to provide some information, i.e., `Type`, `Program` . This parser parse this input and returns user friendly message or `ProofRules`.

4.4.2 Some Important Functions

There are three very important functions work in the center of our calculus. They are explained below:

checkRule

The definition of this function is

```
checkRule :: Env -> ProofRules -> ProofState -> Bool
```

It takes `environment,ProofRules and ProofState` as argument and checks whether this `ProofRules` is applicable to this `ProofState` or not. If applicable then it returns True otherwise False.

applyRules

The definition of this function is

```
applyRules :: Env -> ProofRules -> ProofState -> (ProofState,
             UndoProofRules)
```

It takes `environement, ProofRules, ProofState` as argument and apply the `ProofRules` to the `ProofState` and finally returns a pair of new `ProofState` and `UndoProofRules`.

undoRules

The definition of this function is

```
undoRules ::UndoProofRules -> ProofState -> ProofState
```

This function takes `UndoProofRules`, `ProofState` as a parameter and after applying `UndoProofRules` to present `ProofState` it returns the previous `ProofState`.

4.4.3 Some Help-Functions

There are some other helping functions that assist previous three important functions. But they are similar to the functions in type module. So here I am just listing them.

- `programSubstitute :: Program -> String -> Program -> Program`
- `progTypeSubstitute :: Type -> String -> Program -> Program`
- `freeProgramVars :: Program -> S.Set String`
- `freeProgramTypeVars :: Program -> S.Set String`
- `programEqual :: Env -> Program -> Program -> Bool`

4.5 Toolkit Implementation

GTK+, Glade and gtk2hs help out the implementation of the toolkit. GTK+ is a toolkit for creating graphical user interface. GTK+ is written in C, but has bindings to many other popular programming languages such as Haskell, C++, Python and among others. GTK+ is licensed under the GNU LGPL 2.1 allowing development of open software, free software, or even commercial non-free software without any license fees or royalties. Check out the latest stable release of GTK+ for GNU/Linux and Unix, Windows(32-bit) and 64-bit or OSX at http://www.gtk.org/download.html [11].

Glade is a RAD tool to enable quick and easy development of user interface for the GTK+ toolkit and the GNOME desktop environment. The user interface designed in Glade are saved as XML. By using the GtkBuilder (GTK+), XML can be loaded dynamically as needed by applications. By using GtkBuilder, Glade XML files can be used in numerous programming languages including Haskell, C, C++, Vala, Java, Perl, Python,and others. Glade is Free Software released under the GNU

GPL License. To get the sources for the Glade project choose one of the release tarballs from http://glade.gnome.org/sources.html [12].

Gtk2Hs is a Haskell binding to Gtk+ 2.x. Using it, one can write Gtk+ based applications with GHC. It currently works with Gtk+ 2.0 through to 2.22 on Unix, Win32 and MacOS X. For installing gtk2hs, haskell platform and the GTK/Glade bundle installation is required. For sources, installation notes and further study visit http://www.haskell.org/haskellwiki/Gtk2Hs#What_is_it.3F [13].

4.6 User Manual

Using our toolkit is really simple and easy. We need to follow just a couple of steps. First of all we will have to write some program in any text editor and save the file. Now we need to run the system and select the file. As a part of the first step we also need to compile the program and if everything goes alright we will have to enter the second step by opening the proof window. In proof window our first job is to give input the property of program we are going to proof and parse it. The final step is to prove the property by applying different rules. It is not necessary to finish the proof in one sitting. If one likes she can save the proof and come later. In order to prove a saved proof, she will have to load the program, parse it, open proof window and load the saved proof sequentially. Now it is open to apply any rule to finish this proof. In the last chapter we will show the precise use of our toolkit with an example. So here we are just explaining functionality of all the components in our toolkit.

4.6.1 Buttons

Select File

This button works as a file selector. When someone click on this button, it will pop-up a new window and ask the user to select a file, i.e., the program whose properties user going to proof.

Accept

Accept button parse the input program (contents of the selected file) and gives appropriate message to the user, either accepted or user friendly error message.

start Proof

This button creates the Proof window.

Insert 1st Proof State

It parses the `Theorem` provided in the text input box (by the left of this button), and gives appropriate message to the user, either accepted or user friendly error message.

Help/Output

It provides general help message about every action.

Show Proof

It will display the proof, i.e. the list of `ProofRules` applied, into another pop-up window. If user wants to close this new window, needs to press the close button or cross sign at the top right hand.

Undo & Redo

This two buttons are straight forward. If applicable they will do the undo and redo. On the other hand it will display user friendly error message saying that "undo/redo are not possible".

Save

User does not have to complete a proof in one sitting. User may also return to current position in the future, to refresh their memory or any other purpose. If some one wants to come back later and finish his proof from present state, she needs to save her work. If user click on the save button it will create a pop up window and ask for the file where she wants to save her present state.

Load

It loads the saved proof, `ProofState` and `Theorem`. Therefore if user click on this button, it creates a pop up window and ask for the file name that contain her saved work.

Input Hint

It directs the user toward the right input by displaying messages, i.e. `Program` or `Type` required, input is wrong etc.

Go

Go button parse the user input. Depending on the parser output user will be able to apply the current rule or get an user friendly error message.

4.6.2 List of Proof Button

Each proof button is associated to one `ProofRules`, explained in chapter two. The list of proof buttons are as follows:

Table 4.3: Proof Buttons

Type of Rule	Name of Rule	Type of Rule	Name of Rule
Basic Rules	Symmetry Transitivity Subsumption Declaration Swap State	Record Rules	Declaration Selection Beta(Selection) Update Beta(Update)
Function Rules	Declaration application Beta-Rule	Extended Record Rules	Declaration Beta-Rule
Object Rules	Declaration Selection Beta(Selection) Update Beta(Update)	Polymorphic Rules	Declaration application Beta-Rule

4.7 Example

In this section we are going to prove some properties of `Program` written in our language. As a first step of this process, in any text editor we need to write some `Program` and save the file. For this example the name of the file is FirstProg.prog. In order to start the system we need to run (double click) main.hs module with GHCi. After a successful run we will see the following window.

Figure 4.1: GHCi command window

To open the first window of our toolkit we need to write down the command `main` in GHCi command prompt (present window). The first window of our toolkit is as follows:

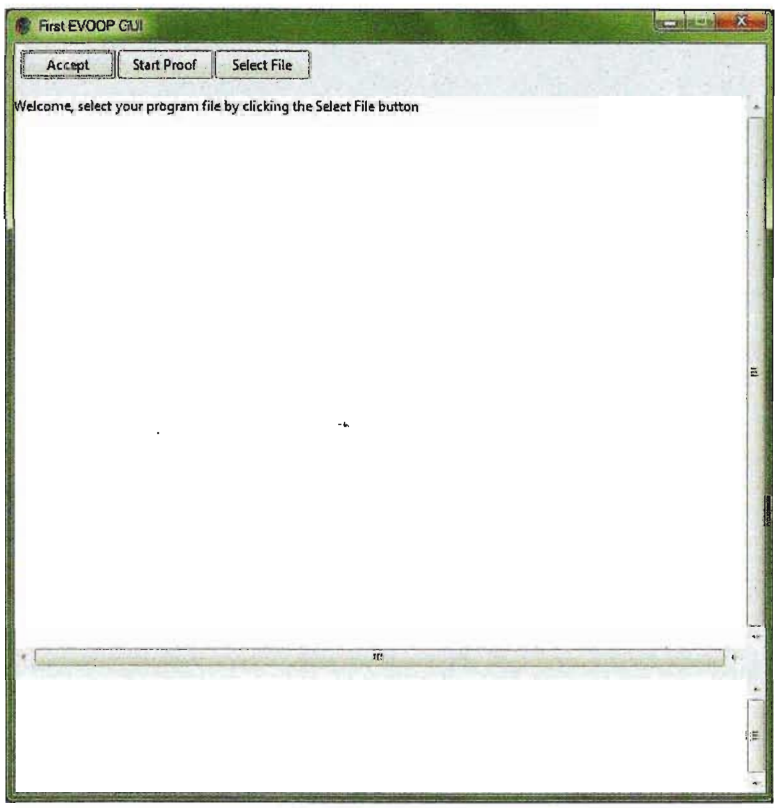

Figure 4.2: First VOOP window

Next job is to select the file FirstProg.prog by clicking the `Select File` button. In order to parse the file we will have to click the `Accept` button. If there is no error in the `Program`, the message `Accepted` otherwise user friendly error message will show up on bottom text view box. The snap short of the accepted program is as follows:

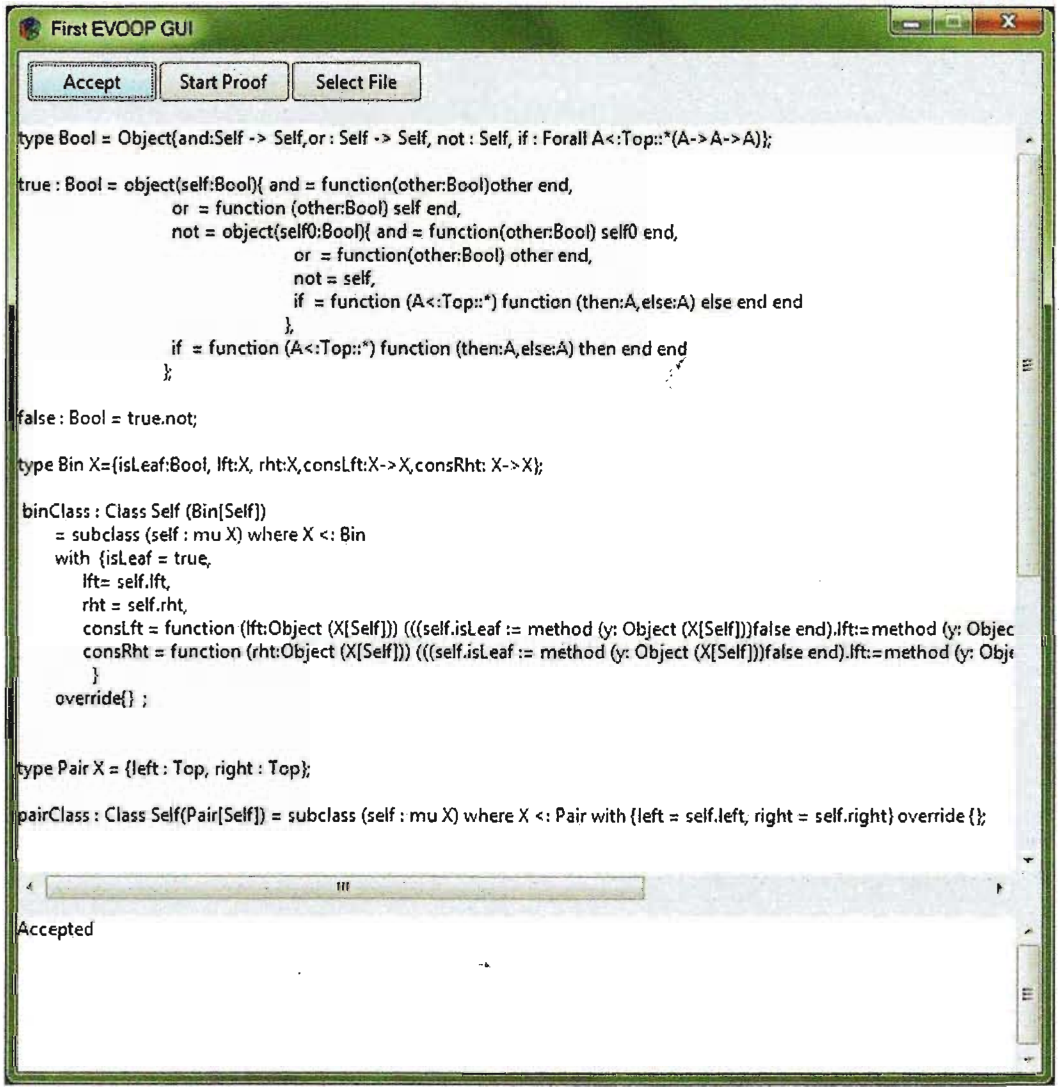

Figure 4.3: First VOOP after parsing the Program

Now it is time to open the proof window by clicking the `Start Proof` button. The view of the proof window is as follows:

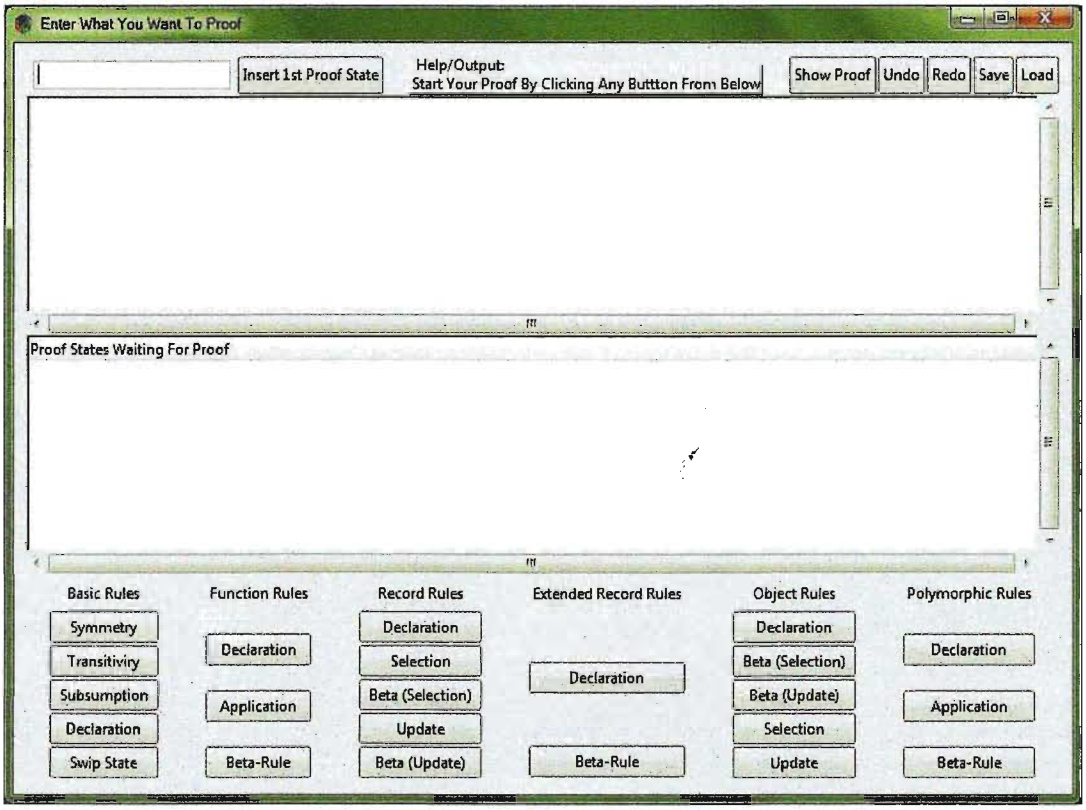

Figure 4.4: Proof window

Step 1:

We want to prove that the implementation of `Program or` is correct for all conditions. At this point we need to select the property of `Program` we are going to prove. We are familiar with all the three properties of `or`, i.e, **TRUE OR X = TRUE, FALSE OR FALSE = FALSE**. In order to start the proof we have to input the first theorem, i.e., the property of the `Program` we are going to prove, in the input text box. In this example the theorem we are going to prove is `(true.or)[false]==true:Bool`

Step 2:

Click the `Insert 1st Proof State` button to accelerate the proof by parsing the first theorem. The view of the proof window is as follows:

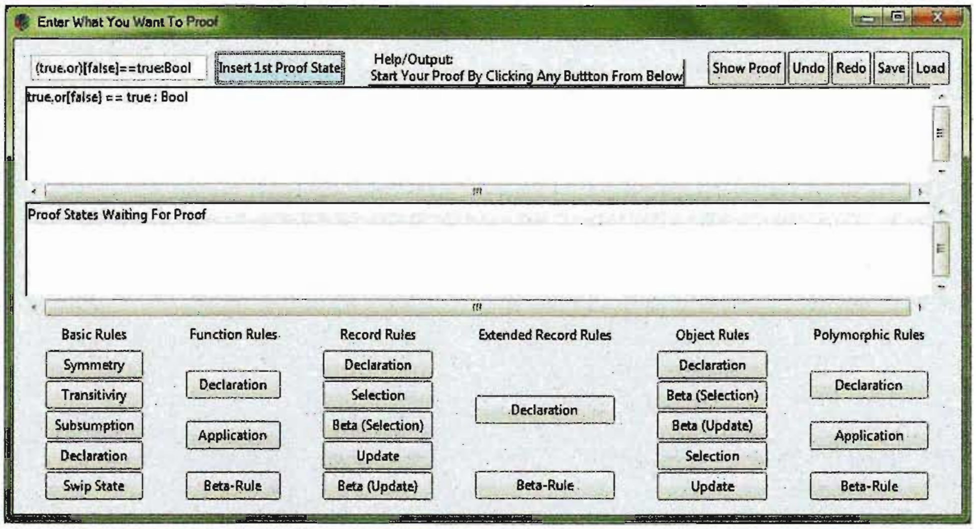

Figure 4.5: Proof window with first theorem

Now we need to apply the following rules sequentially.

Rule 1: Transitivity (Basic Rule)

For the given theorem, this is the first rule we are able to apply. For this rule some user input is required. So at the time of clicking the `Transitivity` button one new window will pop up. we will have to enter `function (other:Bool) true end [false]` into the input text box on new pop up window. In order to parse this user input we will have to press the `Go` button. The snap short of pop up window with the user input is as follows:

Figure 4.6: User input window

After applying the first rule, the proof window looks like as follows:

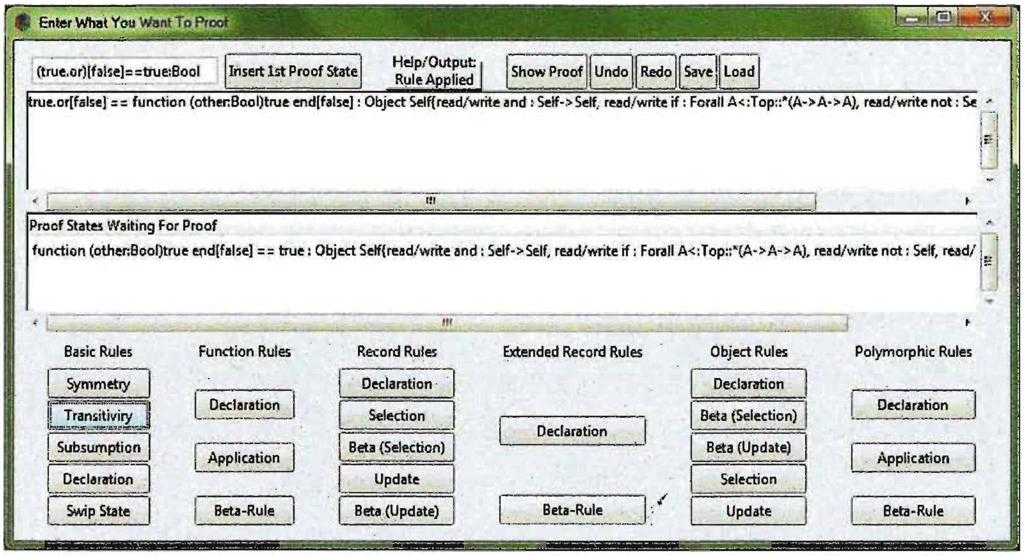

Figure 4.7: Proof window after rule1

Rule 2: Application (Function Rules)

This is the second rule we are going to apply. For this rule we also need an user input. The required user input is `Bool`. The view of the proof window after applying second rule is as follows:

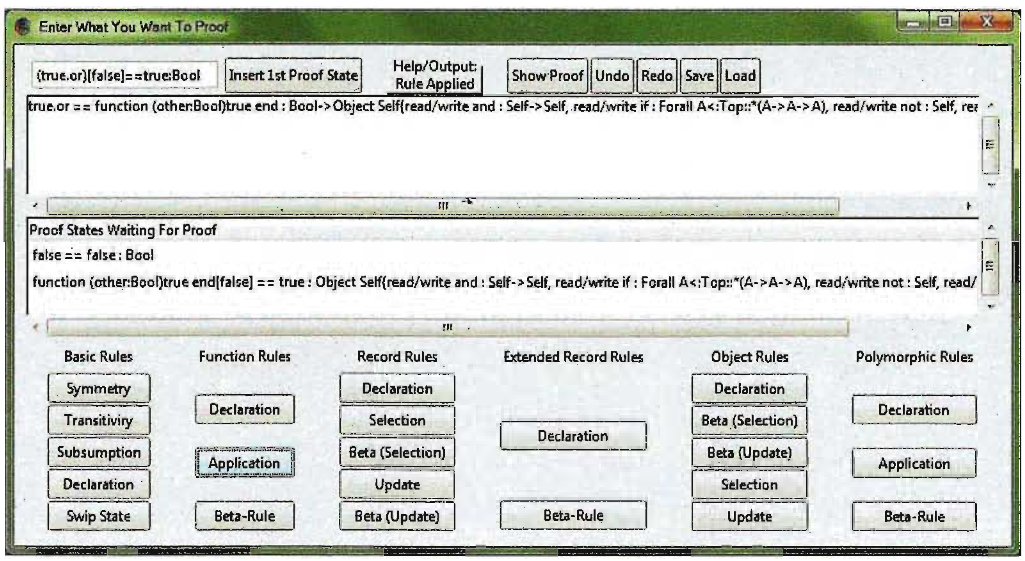

Figure 4.8: Proof window after rule2

Rule 3: Transitivity (Basic Rule)

It is again time to apply the Transitivity rule. As we said for this rule user input is required. The required input for this time is as follows:

```
Required Input:
object(self:Bool){ and = function(other:Bool)other end,
      or  = function (other:Bool) self end,
      not = object(self0:Bool)
      {
      and = function(other:Bool) self0 end,
      or = function(other:Bool) other end,
      not= self,
      if = function (A<:Top::*) function (then:A,else:A) else end end
      },
      if = function (A<:Top::*) function (then:A,else:A) then end end
  }.or
```

The new look of the proof window after applying third rule is as follows:

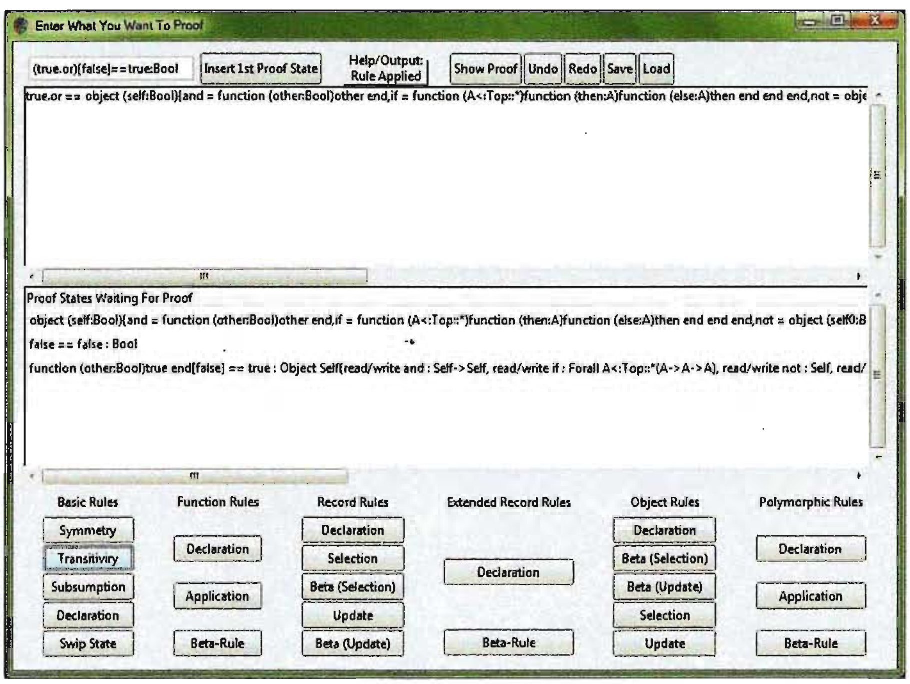

Figure 4.9: Proof window after rule3

Rule 4: Selection (Object Rules)

In order to apply this rule, the following user input is required.

```
Required Input:
Object{and:Self -> Self,or : Self -> Self,
not : Self,if : Forall A<:Top::*(A->A->A)};
```

The following image describes the proof window after applying the fourth rule.

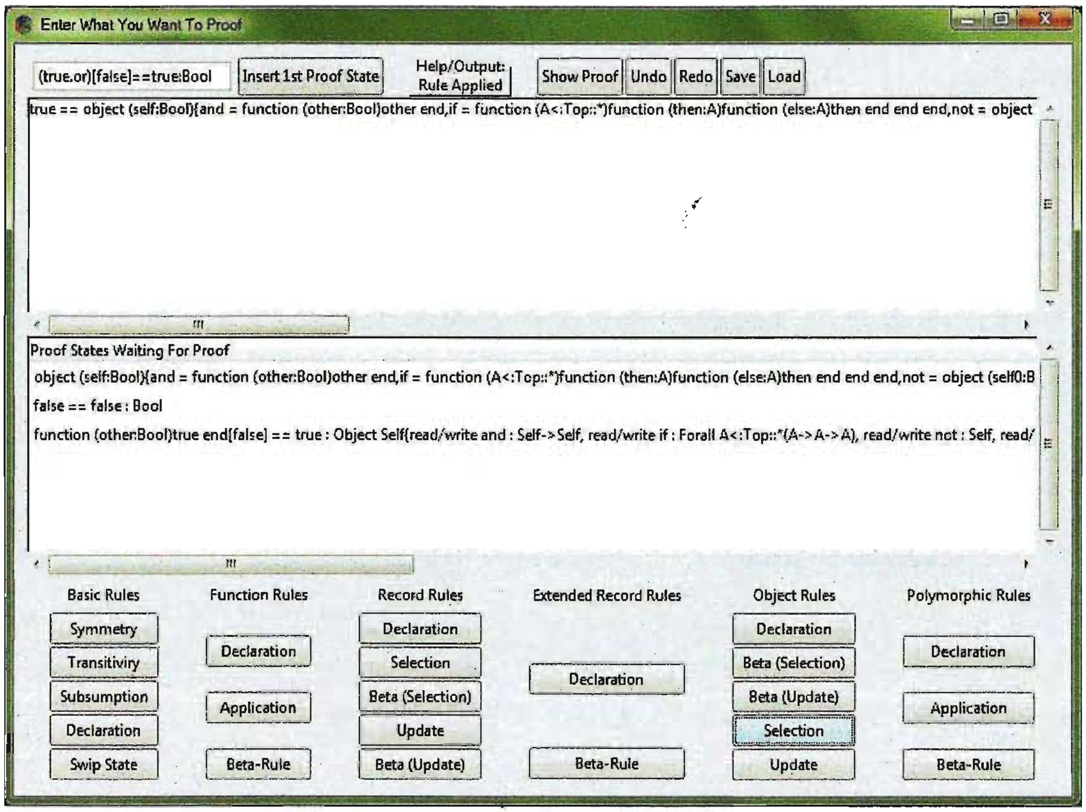

Figure 4.10: Proof window after rule4

Rule 5: Declaration (Basic Rules)

For this rule user input is not required. The view of the proof window after applying this rule is as follows:

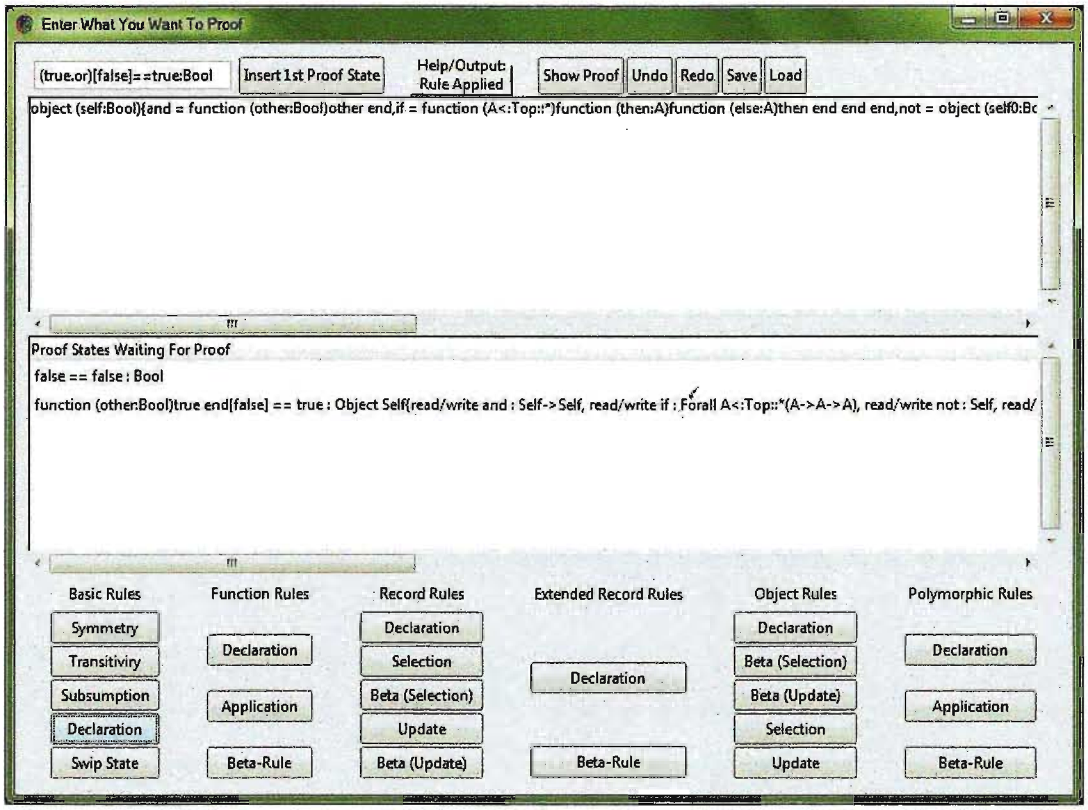

Figure 4.11: Proof window after rule5

Rule 6: Beta Selection (Object Rules)

No user input is required for this rule. The proof window after applying this rule is printed below.

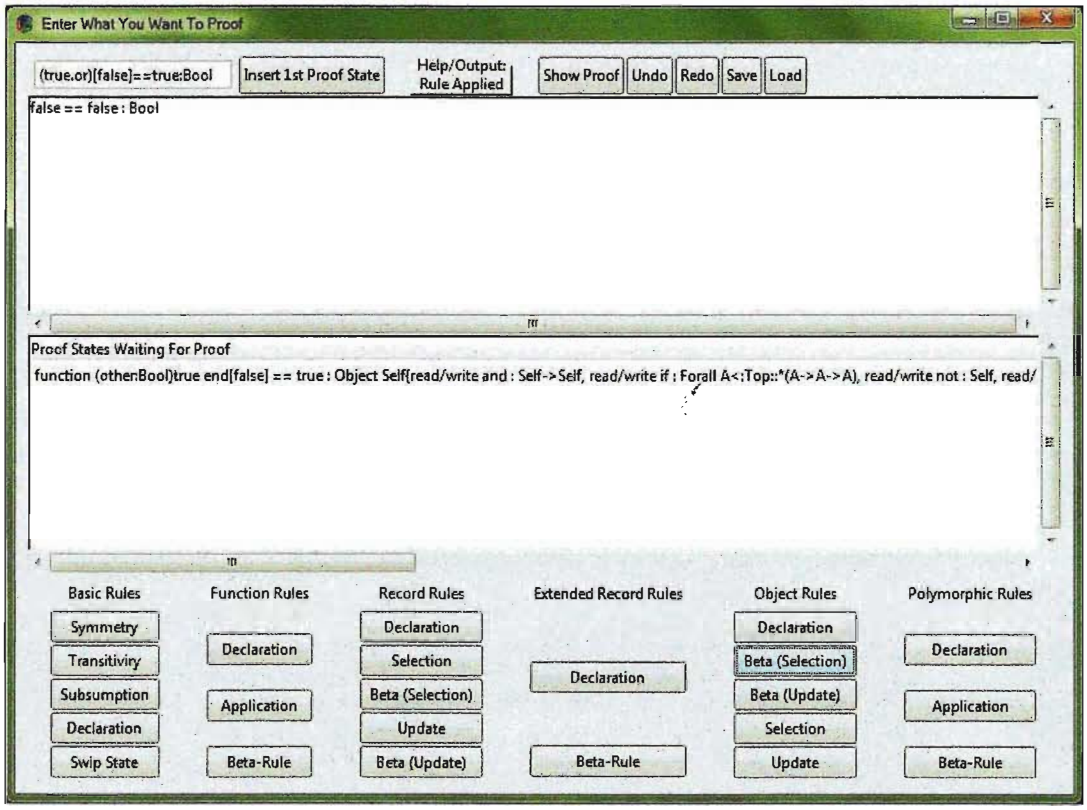

Figure 4.12: Proof window after rule6

Rule 7: Declaration (Basic Rules)

As we stated before, for this rule user input is not required. So after applying this rule the proof window will look like the picture below.

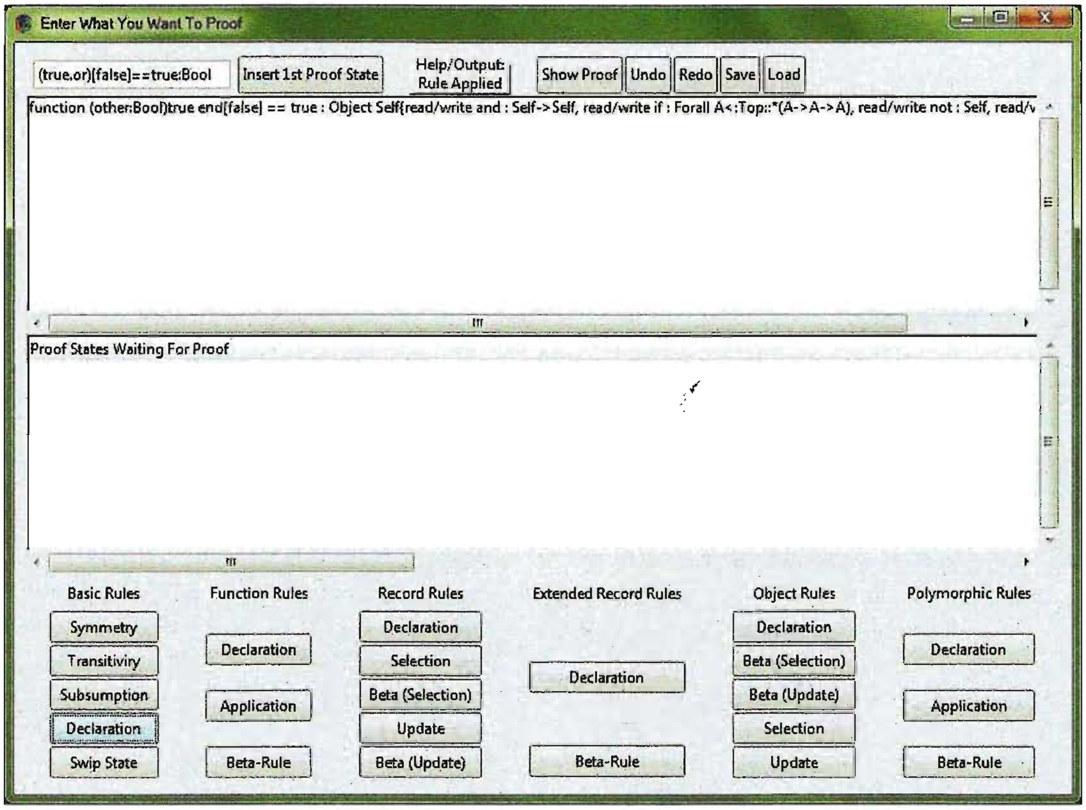

Figure 4.13: Proof window after rule7

Rule 8: Beta-Rule (Function Rules)

In order to apply this rule we need not to provide any user input. In order to prove the theorem, this is the last rule we are going to apply. So the final view of the proof window is as follows:

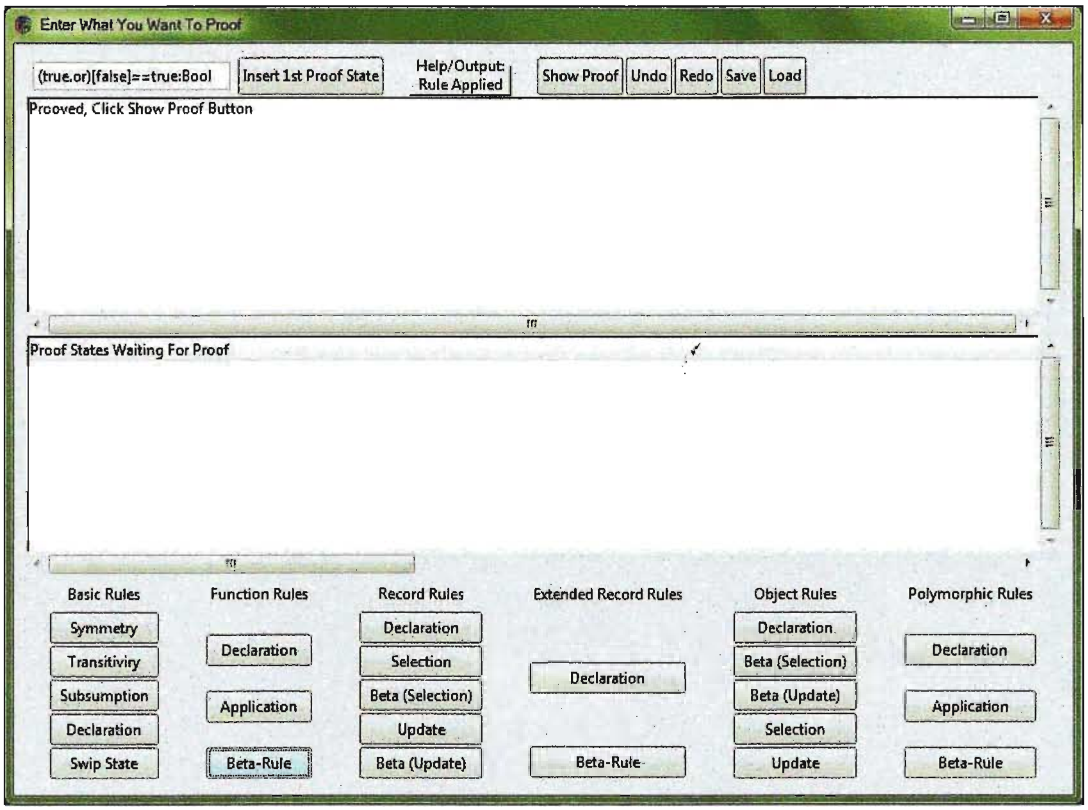

Figure 4.14: Proof window after rule8

So if we prove two other properties, we will be able to insure that program or will do what it was intended to do.

Chapter 5

Conclusion and Future work

5.1 Conclusion

There is no question about the importance of proving correctness of an implementation in a lot of applications. In order to make such an ambitious task as convenient as possible we have defined and implemented a powerful object oriented programming language. The syntax of this language is specified explicitly with examples. The calculus also took an important part of our concern. We explained the calculus in detail and enrich it to a pinnacle. The toolkit is user friendly and easy to use. Finally by proving a properly of one of our programs using our toolkit we indicated that proving correctness of object oriented programs is really feasible in practice.

5.2 Future Works

There are a lot of opportunities for further work on our system. Several of them are stated below.

- Though the present program syntax is user friendly, there is still room for improvement. Currently we are representing higher order matching by using operator type. But one can explicitly use the higher order matching in the programming language instead of operators and make the subclassing more convenient. Adding this would require to extend the syntax of the language by this new kind of matching between object types and to enrich the system of rules by appropriate ones based on the interpretation using type operators.

- In the beginning of our example, we are using the transitivity and the function application rule in order to expand `true.or` by the definition of true and the selection rule. Similar applications of transitivity combined with other rules are used later in order to apply certain rules to subprograms. It would be nice if one could apply the corresponding rules immediately to those subprograms. Future work could focus on this aspect by either adding such a feature to the system or by allowing the user to define some kind of macros on the proof level.

- A rich library of verified programs should be developed. In order to manage those libraries and to make the access as easy as possible the language has to be extended by some kind of module system. This module system should be hierarchical with proper import and export mechanisms.

- Currently it is not possible to execute a program. One way of achieving this is to translate every program into the syntax of a common programming language such as C++. Beside a careful translation such a project would also require to verify that correct programs are translated into correct C++ programs.

Bibliography

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] "Object-oriented programming." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 22 July 2004. Web. 20 May 2011.
http://en.wikipedia.org/wiki/Object-oriented-programming.

[3] Kim B. Bruce. A paradigmatic object-oriented programming language: Design, static typing and semantics. *Journal of Functional Programming*, 4:127–206, 1993.

[4] Kim B. Bruce, Angela Schuett, Robert van Gent, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.

[5] Luca Cardelli. On subtyping and matching. In *In Proceedings ECOOP '95*, pages 145–167. Springer-Verlag, 1995.

[6] William Cook. A proposal for making eiffel type-safe. In *The Computer Journal*, pages 57–70. Cambridge University Press, 1989.

[7] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 125–135, NY, USA, 1990. ACM.

[8] Bertrand Meyer. *Eiffel: the language.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[9] Bruce, K.B., Cardelli, L.,Castagna, G.,The Hopkind Object Group, Leavens, G.T., and Pierce,B. On binary methods. Theory and Practie of Object Systems 1(3), pages 217–238,John Wiley & Sons Canada, Limited, 1995.

[10] "Haskell." HaskellWiki, 15 April, 2003. Web. 6 April 2011. http://www.haskell.org/haskellwiki/Haskell

[11] "GTK+." The GTK+ Team. 7 April 2011. http://www.gtk.org/index.php

[12] "Glade- A User Interface Designer." The Glade Project. Web. 5 April 2011. http://glade.gnome.org/

[13] "Gtk2Hs". HaskellWiki,15 April, 2003. Web. 10 April 2011. http://www.haskell.org/haskellwiki/Gtk2Hs#What_is_it.3F

[14] "Parsec." HaskellWiki,15 April, 2003. Web. 10 April, 2011. http://www.haskell.org/haskellwiki/Parsec

[15] "Free variables and bound variables." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc 22 July 2004. Web. 23 June 2011. http://en.wikipedia.org/wiki/Free_variables_and_bound_variables

[16] Ian Barland, John Greiner, Phokion Kolaitis, Matthias Felleisen, Moshe Vardi. "First-Order Logic: bound variables, free variables." Connexions: a registered trademark of Rice University. Web. 11 June. 2011. http://cnx.org/content/m12081/latest/

[17] "Substitution." Wikipedia: The Free Encyclopedia. Wikimedia Foundation, Inc. 22 July 2004. Web. 8 March 2011. http://en.wikipedia.org/wiki/Substitution_(logic)

[18] Simon Thompson. Haskell The Craft of Functional Proa ramming Second edition.

[19] http://en.wikipedia.org/wiki/Formal_specification

[20] http://en.wikipedia.org/wiki/Formal_verification

[21] http://en.wikipedia.org/wiki/Formal_methods

[22] C. A. R. Hoare. "An axiomatic basis for computer programming". Communications of the ACM, 12(10):576-580,583 October 1969.

[23] R. W. Floyd. "Assigning meanings to programs." Proceedings of the American Mathematical Society Symposia on Applied Mathematics. Vol. 19, pp. 19-31. 1967.

[24] http://hol.sourceforge.net/

[25] http://en.wikipedia.org/wiki/HOL_theorem_prover

[26] http://www.cs.utexas.edu/users/moore/acl2/

[27] http://en.wikipedia.org/wiki/ACL2

[28] http://coq.inria.fr/

[29] http://coq.inria.fr/cocorico/

[30] http://gtps.math.cmu.edu/tps.html

[31] http://pvs.csl.sri.com/

[32] http://www.mizar.org/

[33] http://www.lama.univ-savoie.fr/ RAFFALLI/phox.html

[34] http://en.wikipedia.org/wiki/Interactive_theorem_proving